Erkki Mäkinen (toim.)

Tietojenkäsittelytieteellisiä tutkielmia
Talvi 2016

INFORMAATIOTIETEIDEN YKSIKKÖ
TAMPEREEN YLIOPISTO

INFORMAATIOTIETEIDEN YKSIKÖN RAPORTTEJA 43/2016

TAMPERE 2016

TAMPEREEN YLIOPISTO
INFORMAATIOTIETEIDEN YKSIKKÖ
INFORMAATIOTIETEIDEN YKSIKÖN RAPORTTEJA 43/2016
MAALISKUU 2016

Erkki Mäkinen (toim.)

Tietojenkäsittelytieteellisiä tutkielmia
Talvi 2016

INFORMAATIOTIETEIDEN YKSIKKÖ
33014 TAMPEREEN YLIOPISTO

ISBN 978-952-03-0092-0 (pdf)

ISSN-L 1799-8158
ISSN 1799-8158

Sisällysluettelo

Takaisinkytkeytyvät neuroverkot

Toni Helenius

Tiivistelmä

Takaisinkytkeytyvät neuroverkot ovat tarkoitettu sekvenssimuotoisen tiedon mallintamiseen, jossa tieto on toisistaan riippuvaista. Tällaista tietoa on esimerkiksi luonnollinen kieli, jossa sanojen merkitys riippuu aiemmin annetuista. Takaisinkytkeytyvien neuroverkkojen mallinnusvoima kuitenkin rajoittuu hyvin lyhyisiin sekvensseihin. Ongelmaa on pyritty ratkaisemaan kehittämällä edelleen LSTM-verkot, jotka ovat erikoistuneet pidempiin sekvensseihin ja riippuvaisuuksiin. Tässä tutkielmassa esitellään takaisinkytkeytyvien neuroverkkojen ja LSTM-verkkojen minimaalinen toteutus.

Avainsanat: Takaisinkytkeytyvät neuroverkot, LSTM

1 Johdanto

Keinotekoiset neuroverkot ovat alun perin aivojen hermoverkkojen toimintatavan inspiroima laskentamalli. Vaikka neuroverkkojen kehitys voidaan sanoa alkaneen jo 1940-luvulta, se on vasta viime vuosikymmenen aikana noussut suureen suosioon. Puheentunnistus, hakukoneet ja käännöstyökalut ovat muutamia kuluttajia lähellä olevia koneoppimisen ja neuroverkkojen sovelluksia.

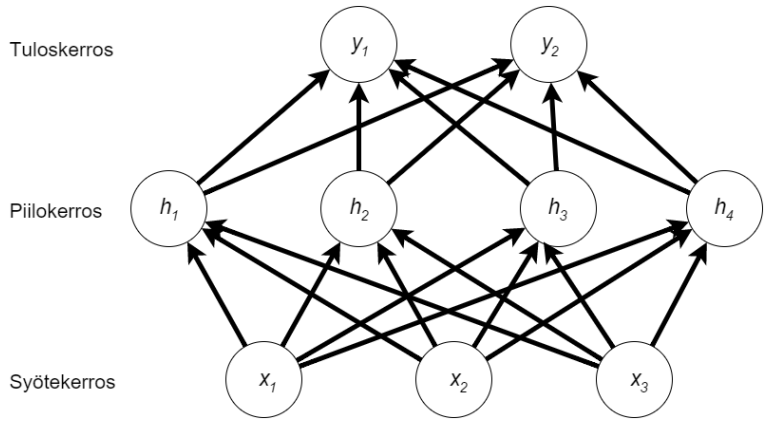
Neuroverkkojen teorian kehittyminen yhdessä tietokoneiden laskentatehon ja tiedon määrän kasvun kanssa on tehnyt mahdolliseksi hyvinkin tehokkaat luokittelu-, likiarvo- ja prosessointifunktiot. Suuret neuroverkot vaativat yhä huomattavia opetusaikoja, mutta valmista neuroverkkoa on laskennallisesti kevyt käyttää, ja esimerkiksi Google Translate-nimisessä visuaalisessa kääntäjässä käyttäjä hyödyntää puhelimessa olevaa neuroverkkoa. Tällöin kuvaa ei tarvitse siirtää Googlen palvelimille käännettäväksi ja käännöstyö sujuu melko nopeasti, käyttäjän Internet-yhteyttä käyttämättä. (Good, 2015)

Takaisinkytkeytyvät neuroverkot sopivat erityisesti sekvenssimuotoisen tiedon luokitteluun. (Sutskever, 2013) Tällaisia ovat muun muassa luonnolliseen kieleen, puheentunnistukseen tai kääntämiseen liittyvät tehtävät. Kerron tässä tutkielmassa aluksi yleisesti neuroverkoista ja yksinkertaisemmista eteenpäinsyöttävistä neuroverkoista. Sen jälkeen esittelen takaisinkytkeytyvät neuroverkot ja niistä kehitetyt LSTM-verkot.

2 Neuroverkot

Keinotekoiset neuroverkot ovat yksi koneoppimisen (machine learning) tekniikoista, jolla pyritään automaattiseen piirteenirroitukseen käyttämällä ns. piirreoppimista (feature learning). Koneoppimisessa esimerkkitiedosta pyritään muodostamaan malleja ja yleistämään sääntöjä havainnoimalla tiedosta löytyviä piirteitä. Piirteet (features) ovat mitattavia ominaisuuksia, joita tiedosta voi havaita. Esimerkiksi käsinkirjoitettuja numeroita luettaessa piirteet voivat olla tietynlaisia kaaria tai viivoja, joita yhdistelemällä numerot muodostuvat. Puheentunnistuksessa piirteet voivat olla vaikkapa foneemeja (phoneme), jotka vastaavat puhutun kielen eri äänteitä. Perinteisessä koneoppimisessa tällaisia piirteitä on pyritty irroittamaan laatimalla piirteenirroittajia (feature engineering). Tämä on kuitenkin osoittautunut työmäärältä vaativaksi tehtäväksi (Lecun et al., 2015). Siksi apua on lähdetty hakemaan keinotekoisista neuroverkoista, joilla tiedon piirteet pyritään tunnistamaan automaattisesti (feature learning).

Neuroverkot koostuvat yksiköistä (units) ja painokertoimista (weighted connections). Painokertoimia siirtämällä pyritään tuottamaan neuroverkosta approksimaatiofunktio, joka tuottaa haluttuja tuloksia syötteitä annettaessa. Yksinkertaisimmillaan neuroverkot voidaan esittää kerroksina yksiköitä ja niiden välisiä painotettuja yhteyksiä, kuten kuvassa 1. Syötekerroksen yksiköt ovat yhteydessä piilokerroksen yksiköihin, jotka ovat yhteydessä tuloskerroksen yksiköihin. Tällaisia neuroverkkoja kutsutaan eteenpäinsyöttäviksi neuroverkoiksi.

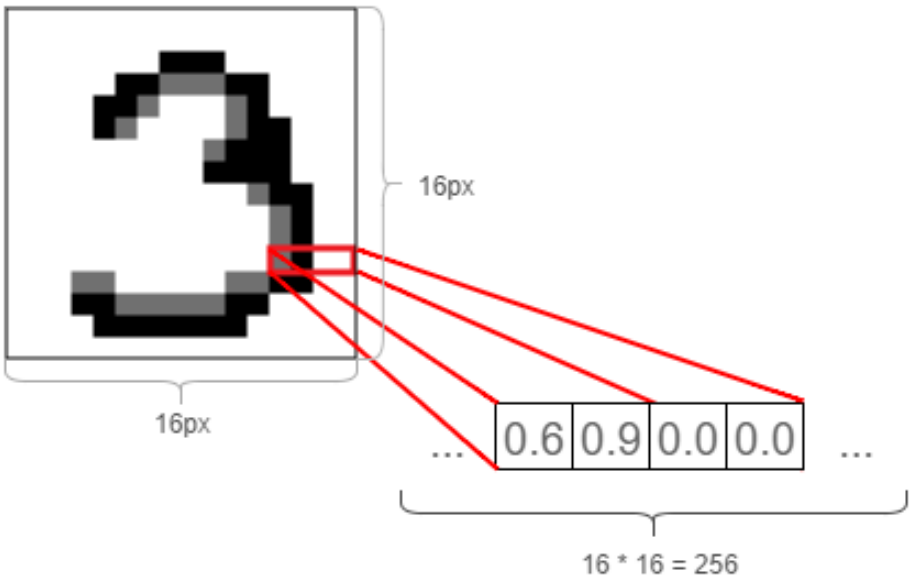


Kuva 1: Eteenpäinsyöttävän neuroverkon yksiköt ja painotetut yhteydet esitettynä kerroksina.

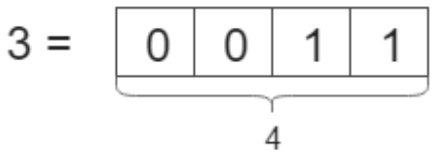
2.1 Neuroverkon opettaminen

Yleisin tapa opettaa neuroverkkoa on ohjattu oppiminen (supervised learning). Silloin neuroverkkoa opetetaan syöte-tavoite-pareilla eli opetusesimer-

keillä (training examples). Opetusesimerkkien syötteet ja tavoitteet annetaan vektoreina. Esimerkiksi käsinkirjoitettuja numeroita lukevaa neuroverkkoa opettaessa syötteet ovat kuvan sisältämien pikselien pituinen vektori, jossa kutakin väriä edustaa vastaavassa kohdassa oleva harmaasävy. Tämä on esitettynä kuvassa 2. Tavoitteena voisi olla esitettynä numeroa esittävien bittien jono vektorina, kuten kuvassa 3 on esitetty.



Kuva 2: Käsinkirjoitetun numeron 16×16 kuvan muuttaminen 256 pituiseksi harmaasävyjen vektoriksi.



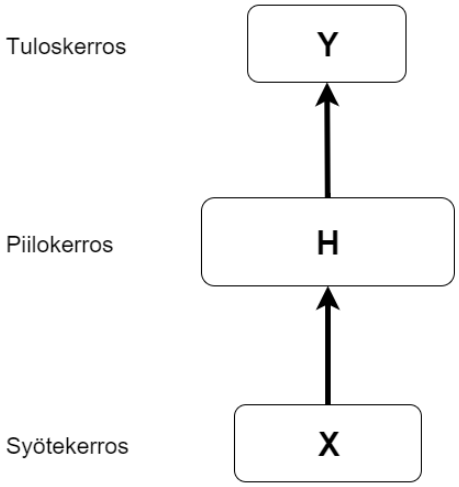
Kuva 3: Kuvaa 2 vastaava numero bittimuodossa neljän pituisessa vektorissa.

Eteenpäinsyöttävää neuroverkkoa opettaessa sen syötteelle antamaa tulosta verrataan opetusesimerkin tavoitteeseen. Tulosta ja tavoitetta vertaamalla saadaan tieto siitä, millä tavoin neuroverkon ennuste oli virheellinen. Painokertoimia muuttamalla tätä ennustetta pyritään muuttamaan siten, että seuraavalla kerralla neuroverkon antamat tulokset saadaan lähemmäs tavoitteita. Tämä prosessi toistetaan käyttämällä suurta määrää opetusesimerkkejä.

Neuroverkolle opetusesimerkkiä opettaessa käytetään jakoa kahteen vaiheeseen: Eteenpäin kulkuun (forward pass), jossa neuroverkon syötteelle antama tulos lasketaan, ja taaksepäin kulkuun (backward pass), jossa tavoitteen

ja tuloksen ero eli virhe vastavirrataan (backpropagate) takaisin neuroverkkoon. Jälkimmäisen vaiheen tarkoituksena on neuroverkon painokertoimien muuttaminen.

Esittelen eteenpäinsyöttävien neuroverkkojen opetuksessa käytettävät kaavat yksikkömuotojen lisäksi matriiseja käyttäen. Myöhemmissä osuuksissa laskukaavoissa käytän pelkästään matriiseja, joita käyttämällä neuroverkot useimmiten ohjelmoidaan. Neuroverkkojen syötteistä puhutaan yleensä vektoreina, mutta käsittelen niitä tässä tutkielmassa rivimatriiseina, jotta voin suoraan käyttää niitä matriisituloissa muiden neuroverkon osien kanssa. Eteenpäinsyöttävä neuroverkko on esitetty matriiseina kuvassa 4.



Kuva 4: Eteenpäinsyöttävän neuroverkon kerrokset ja painotetut yhteydet esitettynä matriiseina.

2.2 Eteenpäin kulku

Eteenpäin kulkiessa lasketaan neuroverkon tulos annetulle syötteelle. Kun syötevektori on annettu neuroverkon I levyiselle syötekerrokselle, J levyisen piilokerroksen yksiköt ottavat syötekerroksen yksiköiltä x_i saamansa syötteiden painokertoimilla w_{ij} painotetun summan. Piilokerroksen yksiköiden arvot h_j saadaan kaavalla

$$h_j = \tanh(\sum_{i=1}^{I} w_{ij} x_i), \tag{2.1}$$

jossa epälineaarisena aktivaatiofunktiona käytettävä hyperbolinen tangentti *tanh* määritellään

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}. \tag{2.2}$$

Epälineaarista aktivaatiofunktiota (activation function) käytetään rajoittamaan arvot tietylle välille, tässä tapauksessa välille $[-1, 1]$. Neuroverkon painokertoimet on alustettu ennen opetuksen aloittamista satunnaisesti välille $[-0.1, 0.1]$.

Matriiseja hyödyntäessä neuroverkon kerrokset ja painokertoimet esitetään matriiseina. Tällöin piilokerroksen yksiköiden arvot eli $\mathbf{1} \times \mathbf{J}$ rivimatriisi $\mathbf{H}$ voidaan määrittää syötteiden $\mathbf{1} \times \mathbf{I}$ rivimatriisin $\mathbf{X}$ ja painokertoimien $\mathbf{I} \times \mathbf{J}$ matriisin $\mathbf{W_x}$ matriisitulolla

$$\mathbf{H} = \tanh(\mathbf{X}\mathbf{W_x}), \tag{2.3}$$

jossa aktivaatiofunktio *tanh* sovelletaan matriisiin elementtikohtaisesti. Termillä $\mathbf{W_x}$ tarkoitetaan syötekerroksen ja piilokerroksen välisten painokerrointen matriisia.

Jälleen K levyisen tuloskerroksen yksiköiden arvoja y_k laskettaessa otetaan J levyiseltä piilokerrokselta saatujen arvojen painotettu summa, ja sovelletaan siihen aktivaatiofunktiota:

$$y_k = \tanh(\sum_{j=1}^{J} w_{jk} h_j). \tag{2.4}$$

Matriisioperaatioita käyttämällä $\mathbf{1} \times \mathbf{K}$ kokoinen tuloskerros $\mathbf{Y}$ saadaan $\mathbf{1} \times \mathbf{J}$ kokoisen $\mathbf{H}$ ja $\mathbf{J} \times \mathbf{K}$ kokoisen painokertoimien $\mathbf{W_y}$ matriisitulolla

$$\mathbf{Y} = \tanh(\mathbf{H}\mathbf{W_y}), \tag{2.5}$$

jossa $\mathbf{W_y}$ tarkoittaa piilokerroksen ja tuloskerroksen välisten painokertoimien matriisia.

Tuloskerroksen yksiköiden arvot muodostavat neuroverkon tuloksen, jota verrataan opetusesimerkin tavoitteeseen. Neuroverkon tuloksen ja tavoitteen erotusta hyödynnetään opetuksen seuraavassa vaiheessa.

2.3 Taaksepäin kulku

Kun tavoitteen ja tuloksen erotus eli virhe on laskettu, lähdetään sitä minimoimaan käyttämällä backpropagation- eli vastavirta-algoritmia. Vastavirtaus on ohjatussa oppimisessa käytettävä tekniikka, jolla virhe viedään tuloskerrokselta lähtien takaisinpäin neuroverkkoa pitkin, tarkoituksena selvittää kunkin painokertoimen osuus virheeseen. Kun painokertoimen osuus virheeseen on laskettu, voidaan sitä muuttamalla minimoida osuus ja samalla parantaa neuroverkon muodostamaa approksimaatiota.

Vastavirtauksen laskentaan käytetään derivaattojen laskusääntöjä, jotka on selitettänyt muun muassa Graves (2012). Määritellään aluksi aktivaatiofunktion *tanh* derivaatta

$$\tanh'(x) = 1 - \tanh(x)^2. \tag{2.6}$$

Deltalla merkitään yksiköiden ja painokertoimien muutoksia vastavirratessa virhettä aikaisemmille kerroksille. Lasketaan ensin tuloskerroksen yksiköiden y_k muutos ottamalla kunkin yksikön aktivaatiofunktion derivaatan arvo ja kertomalla se yksikköä vastaavan tavoitteen t_k ja tuloksen erotuksella:

$$\Delta y_k = \tanh'(y_k)(t_k - y_k). \tag{2.7}$$

Matriiseilla vastaava saadaan ottamalla tuloksen $\mathbf{Y}$ aktivaatiofunktion derivaatan ja tavoitteen $\mathbf{T}$ ja tuloksen erotuksen elementtikohtainen tulo ($\odot$):

$$\Delta \mathbf{Y} = \tanh'(\mathbf{Y}) \odot (\mathbf{T} - \mathbf{Y}). \tag{2.8}$$

Jatketaan vastavirtausta laskemalla piilokerroksen muutos. Yksikkökohtaisessa laskutavassa otetaan ensin piilokerroksen yksikön arvon h_j aktivaatiofunktion derivaatta $\tanh'(h_j)$. Kertomalla se painotettuun summaan yksikköön yhteydessä olevista tuloskerroksen muutoksista Δy_k saadaan piilokerroksen yksikön muutos

$$\Delta h_j = \tanh'(h_j) \sum_{k=1}^{K} \Delta y_k w_{jk}. \tag{2.9}$$

Matriiseilla esitettynä otetaan piilokerroksen $\mathbf{H}$ aktivaatiofunktion derivaatta ja kerrotaan se elementtikohtaisesti tuloskerroksen muutoksen $\Delta \mathbf{Y}$ ja painokerrointen transpoosin $\mathbf{W}_\mathbf{y}^\intercal$ matriisitulon kanssa:

$$\Delta \mathbf{H} = \tanh'(\mathbf{H}) \odot (\Delta \mathbf{Y} \mathbf{W}_\mathbf{y}^\intercal). \tag{2.10}$$

Viimeisenä ylempiä määrittelyjä käyttäen lasketaan painokertoimien muutokset. Piilokerroksen ja tuloskerroksen välisten painokerrointen w_{jk} muutokset saadaan painokertoimen yhdistämien piilokerroksen yksikön h_j ja tuloskerroksen yksikön muutoksen Δy_k tulolla

$$\Delta w_{jk} = h_j \Delta y_k. \tag{2.11}$$

Matriiseilla sama laskutoimitus saadaan piilokerroksen transpoosin $\mathbf{H}^\intercal$ ja tuloskerroksen muutoksen $\Delta \mathbf{Y}$ matriisitulolla

$$\Delta \mathbf{W}_\mathbf{y} = \mathbf{H}^\intercal \Delta \mathbf{Y}. \tag{2.12}$$

Syötekerrokselta piilokerrokselle menevien painokertoimien muutokset saadaan kaavan (2.11) tapaan kertomalla vastaava syöte ja piilokerroksen yksikön muutos keskenään:

$$\Delta w_{ij} = x_i \Delta h_j. \tag{2.13}$$

Matriisiesityksenä kaavan (2.12) tapaan painokerrointen $\mathbf{W}_\mathbf{x}$ muutos saadaan syötteiden transpoosin ja piilokerroksen muutoksen matriisitulolla

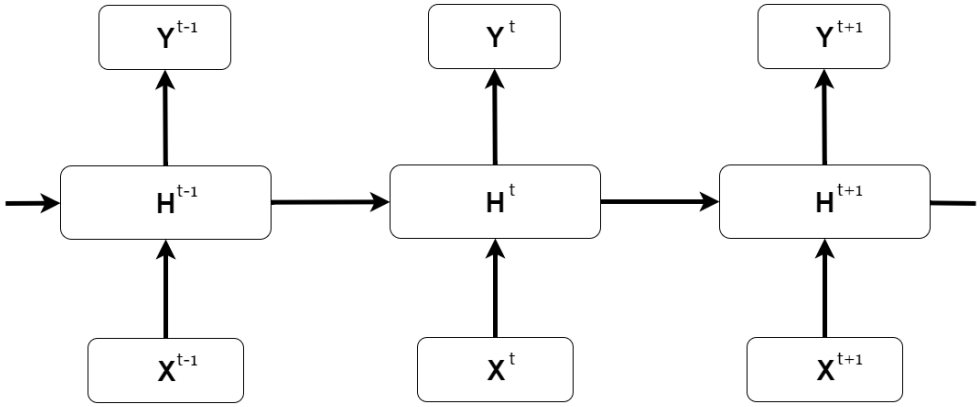
$$\Delta \mathbf{W}_\mathbf{x} = \mathbf{X}^\intercal \Delta \mathbf{H}. \tag{2.14}$$

Painokerrointen muutos $\Delta\mathbf{W_y}$ määrittää lisäyksen painokertoimiin $\mathbf{W_y}$, jolla neuroverkon antamaa virhettä saadaan pienennettyä. Sama suhde on myös painokerrointen muutoksella $\Delta\mathbf{W_x}$ ja painokertoimilla $\mathbf{W_x}$. Nämä lisätään painokertoimiin ja jatketaan seuraavaan opetusesimerkkiin. Tätä neuroverkon opettamisen 'yksi opetusesimerkki kerrallaan' tapaa kutsutaan online menetelmäksi (Graves, 2012).

3 Takaisinkytkeytyvät neuroverkot

Luonnollisen kielen ongelmissa, kuten puheentunnistuksessa ja kielen kääntämisessä, syötteet ovat usein liian suuria ja monimuotoisia suoraan tulkittavaksi. Siksi ne usein pilkotaan sekvensseiksi ääniä, kirjaimia tai sanoja. Nämä sekvenssit sisältävät paljon kontekstiriippuvaista tietoa, jonka avulla esimerkiksi tiettyjen äänteiden tai kirjainten sarja voi tarkoittaa jotain sanaa.

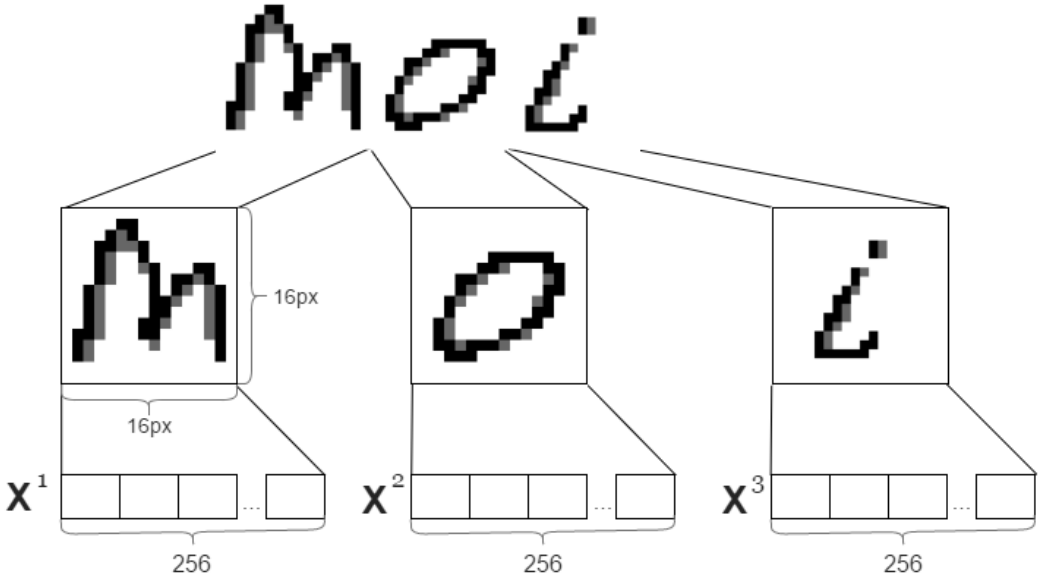
Takaisinkytkeytyvät neuroverkot (recurrent neural networks tai RNN) ovat sekvenssimuotoisen tiedon prosessointiin kehitetty neuroverkkotyyppi. Ne voivat periaatteessa hyödyntää kaikkien aikaisemmin annettujen elementtien historiaa tulosta laskettaessa (Graves, 2012). Takaisinkytkeytyvät neuroverkot oppivat näin kontekstiriippuvaisia merkityksiä syötteille ja syötteistä opituille piirteille.



Kuva 5: Takaisinkytkeytyvä neuroverkko askeleessa t. Piilokerros $\mathbf{H}^t$ ottaa syötteen $\mathbf{X}^t$ lisäksi piilokerroksen aikaisemman arvon $\mathbf{H}^{t-1}$.

Takaisinkytkeytyvät neuroverkot rakentuvat eteenpäinsyöttävien neuroverkkojen pohjalle ja lisäävät painotetun yhteyden piilokerrokselle aikaisemmalta piilokerrokselta. Piilokerrosten välinen yhteys on esitetty kuvassa 5. Takaisinkytkeytyviin neuroverkkoihin syötetään sekvenssit elementti kerrallaan. Hyvä esimerkki voisi olla käsinkirjoitetun tekstin lukeminen. Siinä voidaan hyödyntää eteenpäinsyöttävien neuroverkkojen yksi-yhteen -luokittelukyvyn lisäksi takaisinkytkeytyvien neuroverkkojen kontekstiriippuvaista sekvenssien luokittelua. Teksti syötetään verkolle merkki kerrallaan, jossa ku-

kin merkki on kirjaimen kuva muutettuna harmaasävyjen vektoriksi, kuten kuvan 6 esimerkistä nähdään.



Kuva 6: Käsinkirjoitettu teksti pilkottu merkkeihin, joista syötesekvenssi kootaan.

3.1 Eteenpäin kulku

Takaisinkytkeytyvien neuroverkkojen tuloksen laskeminen aloitetaan samoin kuin eteenpäinsyöttävillä neuroverkoilla, mutta piilokerroksen arvoa $\mathbf{H}^t$ laskettaessa otetaan syötteen arvon askeleessa t lisäksi aikaisemmin saadun piilokerroksen arvo $\mathbf{H}^{t-1}$. Piilokerrosten askelien välisinä painokertoimina käytetään $\mathbf{J} \times \mathbf{J}$ matriisia $\mathbf{W_h}$. Piilokerroksen laskeminen aloitetaan sekvenssin ensimmäisestä elementistä $\mathbf{X}^1$, jolloin $\mathbf{H}^0$ on nollamatriisi $\mathbf{0_{1\times J}}$, kun J on piilokerroksen leveys:

$$\mathbf{H}_t = \tanh(\mathbf{X}^t\mathbf{W_x} + \mathbf{H}^{t-1}\mathbf{W_h}). \tag{3.1}$$

Tuloskerroksen arvo määritellään samoin kuin eteenpäinsyöttävillä neuroverkoilla kaavassa (2.5), mutta sekvenssin askeleessa sijaintia t käyttäen

$$\mathbf{Y}^t = \tanh(\mathbf{H}^t\mathbf{W_y}). \tag{3.2}$$

3.2 Taaksepäin kulku

Takaisinkytkeytyvissä neuroverkoissa käytetään vastavirta-algoritmin muunnelmaa nimeltä backpropagation through time (BPTT) eli aikakeskeinen vastavirtaus (Sutskever, 2013). BPTT:ssä otetaan huomioon neuroverkon se-

kvenssimuotoinen rakenne. Virheet lasketaan askeleittain päinvastaisessa järjestyksessä, aloittaen viimeisestä. Painokertoimet ovat jaettuja kaikkien askeleiden kesken, joten lopuksi painokerrointen muutoksia laskiessa kunkin askeleen virhe summataan yhteen.

Sekvenssin askeleen t tuloksen muutos $\Delta\mathbf{Y}^t$ määritellään kaavan (2.8) tapaan

$$\Delta\mathbf{Y}^t = \tanh'(\mathbf{Y}^t) \odot (\mathbf{T}^t - \mathbf{Y}^t). \tag{3.3}$$

Piilokerroksen muutos $\Delta\mathbf{H}^t$ askeleessa t lasketaan kaavan (2.10) tapaan, mutta ottamalla huomioon piilokerroksen seuraavaksi saadun arvon $\mathbf{H}^{t+1}$ ja painokerrointen transpoosin $\mathbf{W}_\mathbf{h}^\intercal$ tulo:

$$\Delta\mathbf{H}^t = \tanh'(\mathbf{H}^t) \odot (\Delta\mathbf{Y}^t\mathbf{W}_\mathbf{y}^\intercal + \mathbf{H}^{t+1}\mathbf{W}_\mathbf{h}^\intercal). \tag{3.4}$$

Ensimmäistä piilokerroksen muutosta eli $\Delta\mathbf{H}^T$ laskettaessa, $\mathbf{H}^{T+1}$ on nollamatriisi $\mathbf{0}_{\mathbf{1}\times\mathbf{J}}$, kun T on syötesekvenssin pituus ja J piilokerroksen leveys.

Painokerrointen $\mathbf{W}_\mathbf{y}$ muutos askeleessa t lasketaan piilokerroksen transpoosin $\mathbf{H}^{(t)\intercal}$ ja tuloksen muutoksen $\Delta\mathbf{Y}^t$ matriisitulolla. Painokertoimet jaetaan kaikkien askeleiden kesken, joten summataan jokaisesta askeleesta saatu painokerrointen muutos:

$$\Delta\mathbf{W}_\mathbf{y} = \sum_t \mathbf{H}^{(t)\intercal}\Delta\mathbf{Y}^t. \tag{3.5}$$

Takaisinkytkeytyvien neuroverkkojen piilokerroksen askelien välillä on jaetut painokertoimet $\mathbf{W}_\mathbf{h}$. Painokerrointen muutos askeleessa t on seuraavan askeleen piilokerroksen tuloksen transpoosin $\mathbf{H}^{(t+1)\intercal}$ ja piilokerroksen muutoksen $\Delta\mathbf{H}^t$ matriisitulo. Summataan jokaisesta askeleesta saatu piilokerroksen muutos:

$$\Delta\mathbf{W}_\mathbf{h} = \sum_t \mathbf{H}^{(t+1)\intercal}\Delta\mathbf{H}^t. \tag{3.6}$$

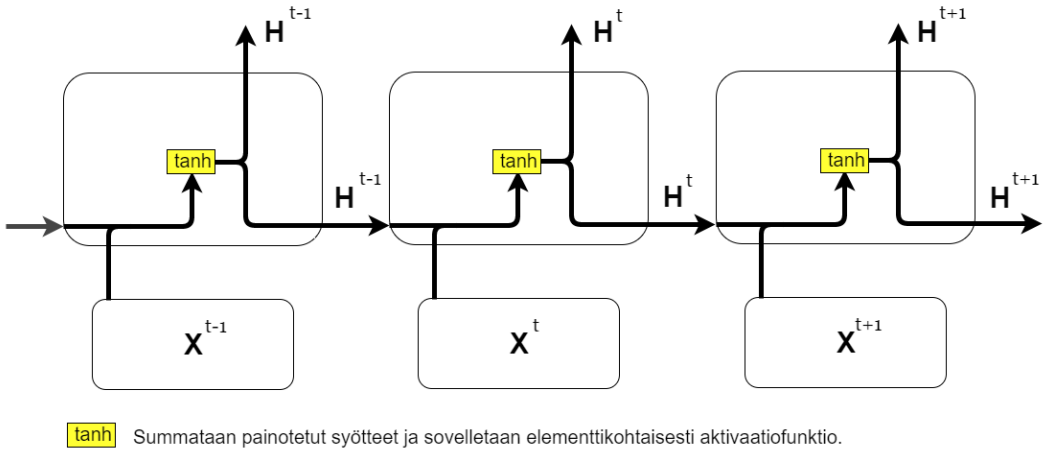
Jälleen kaavan (3.5) tapaan syötekerroksen ja piilokerroksen välisten painokerrointen muutos lasketaan

$$\Delta\mathbf{W}_\mathbf{x} = \sum_t \mathbf{X}^{(t)\intercal}\Delta\mathbf{H}^t. \tag{3.7}$$

Eteenpäinsyöttävien neuroverkkojen tapaan painokerrointen muutokset $\Delta\mathbf{W}_\mathbf{x}$, $\Delta\mathbf{W}_\mathbf{h}$ ja $\Delta\mathbf{W}_\mathbf{y}$ lisäämällä vastaaviin painokertoimiin hivutetaan takaisinkytkeytyvä neuroverkko tarkempaan approksimaatioon, jonka jälkeen jatketaan seuraavaan opetusesimerkkiin.

4 Pitkäkestoinen työmuisti

Takaisinkytkeytyvät neuroverkot on tehty luokittelemaan kontekstiriippuvaista tietoa. Käytännössä on kuitenkin todettu, että varsinkin pidemmillä sekvensseillä ne hukkaavat kontekstin. Kun sekvenssit ovat pitkiä eli verkossa on paljon piilokerrosten askelia, vastavirtaus niiden läpi aiheuttaa virheiden 'räjähtämisen' tai katoamisen. Ongelmaa kutsutaan usein nimellä vanishing gradient ja sitä ovat tutkineet muun muassa Hochreiter et al. (2001). Ongelmaan on tutkittu monia ratkaisuja, joista parhaiten on menestynyt Pitkäkestoinen työmuisti (Long short term memory tai LSTM) (Hochreiter and Schmidhuber, 1995).

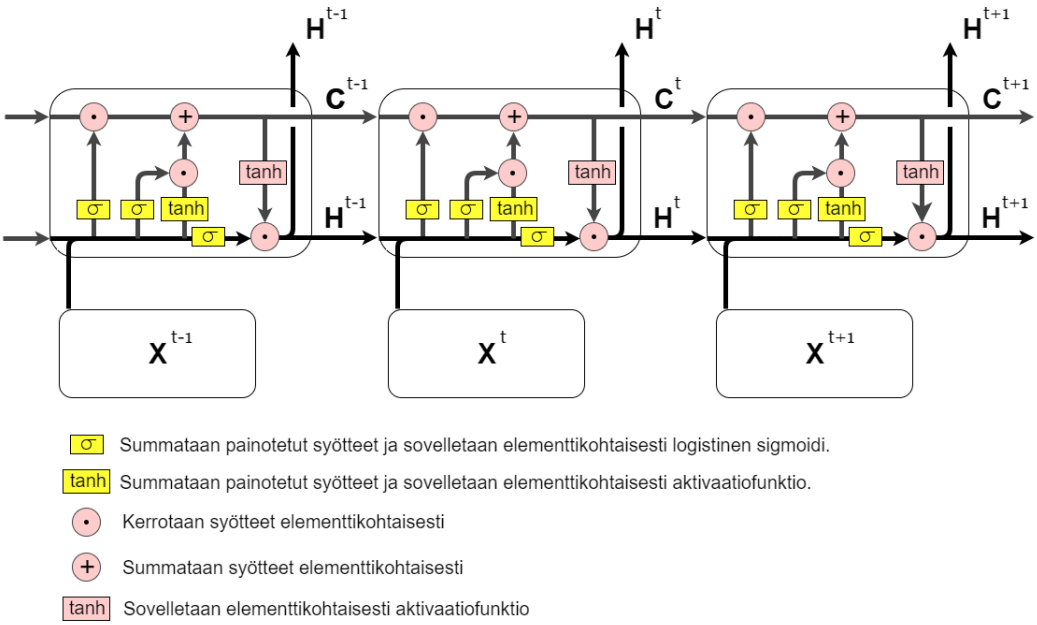


Kuva 7: Tarkempi kuvaus takaisinkytkeytyvän neuroverkon piilokerroksen rakenteesta. $\mathbf{H}^t$ on piilokerrokselta saatu arvo askeleessa t.

Takaisinkytkeytyvien neuroverkkojen piilokerroksella on hyvin yksinkertainen rakenne. Se ottaa aiemman piilokerroksen arvon ja tämänhetkisen syötteen, ja laittaa ne aktivaatiofunktion läpi. Takaisinkytkeytyvän neuroverkon piilokerroksen rakenne on esitetty kuvassa 7. LSTM-verkon piilokerros on monimutkaisempia.

LSTM-verkon piilokerroksessa ylläpidetään erityistä solun tilaa (cell state), joka erikoistuu pidemmän etäisyyden riippuvaisuuksiin sekvenssissä. Tätä muokataan sekvenssin elementtejä syöttäessä käyttämällä unohtamisporttia (forget gate), syöteporttia (input gate) ja tulosporttia (output gate), jotka ovat esitettynä kuvassa 8, ja tarkemmin kuvassa 9. Näitä portteja käyttämällä pyritään säilyttämään solun tilassa tärkeät riippuvaisuussuhteet.

LSTM-verkon kaavat on sovellettu tähän tutkielmaan Greff et al. (2015) paperista.



Kuva 8: Kuvaus LSTM-verkon piilokerroksen rakenteesta, jossa $\mathbf{H}^t$ piilokerrokselta saatu arvo ja $\mathbf{C}^t$ solun arvo askeleessa t.

4.1 Eteenpäin kulku

Lasketaan ensin unohtamisportista saatu arvo kohdassa t. Syötteiden ja unohtamisportin välisinä painokertoimina käytetään $\mathbf{I} \times \mathbf{J}$ matriisia $\mathbf{W_{xf}}$, ja piilokerroksen ja unohtamisportin välisinä painokertoimina $\mathbf{J} \times \mathbf{J}$ matriisia $\mathbf{W_{hf}}$, kun I on syötekerroksen leveys ja J piilokerroksen leveys. Unohtamisportista saatu arvo $\mathbf{F}^t$ saadaan summaamalla syötteen $\mathbf{X}^t$ ja painokerrointen $\mathbf{W_{xf}}$ sekä piilokerroksen aiemman arvon $\mathbf{H}^{t-1}$ ja painokerrointen $\mathbf{W_{hf}}$ matriisitulot:

$$\mathbf{F}^t = \sigma(\mathbf{X}^t\mathbf{W_{xf}} + \mathbf{H}^{t-1}\mathbf{W_{hf}}). \tag{4.1}$$

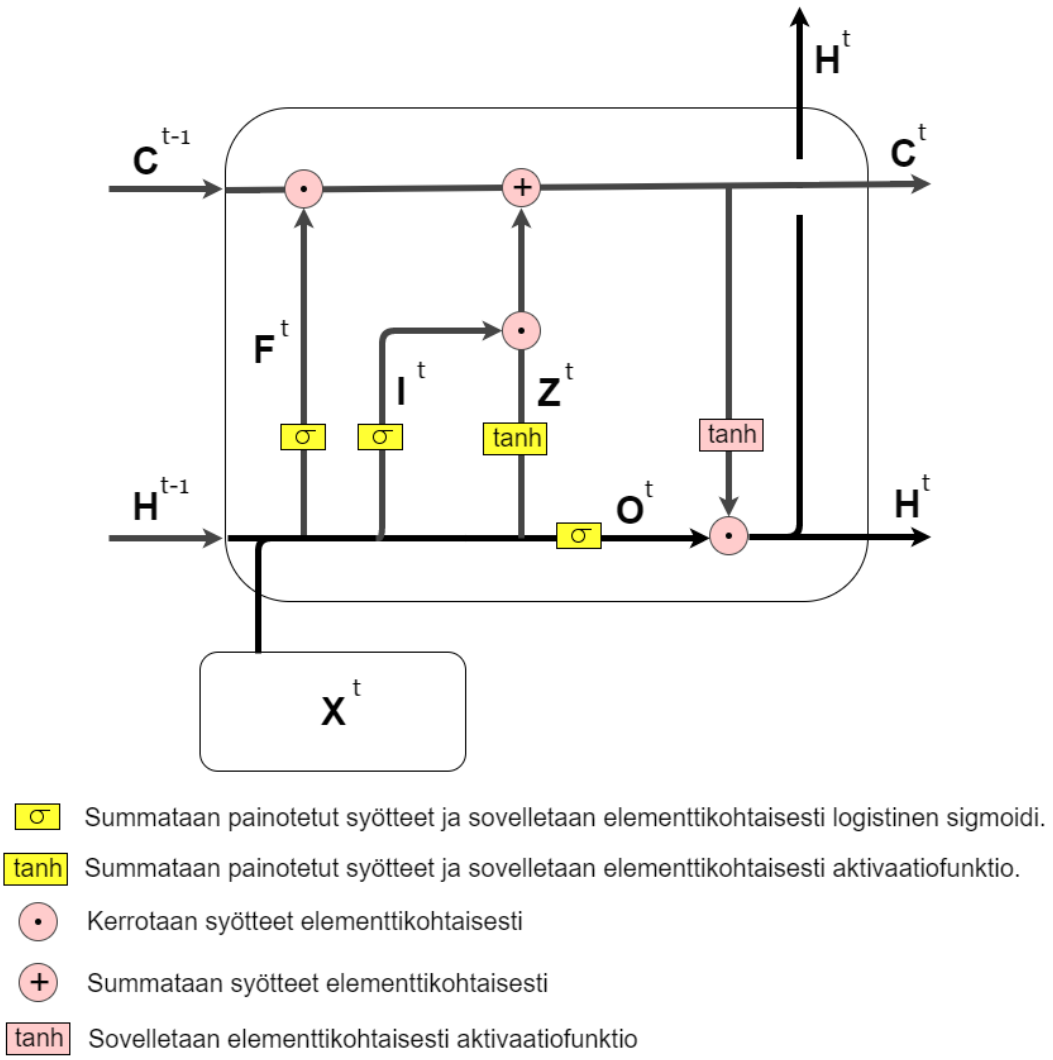
Logistinen sigmoidi σ on määritelty

$$\sigma(x) = \frac{1}{1+e^{-x}}, \tag{4.2}$$

joka sovellettuna matriisiin elementtikohtaisesti sijoittaa arvot välille $[0, 1]$. Logistista sigmoidia käyttämällä LSTM-verkko voi porteilla määritellä säilytettävät ja poistettavat tiedot. Unohtamisportista saatua arvoa $\mathbf{F}^t$ käytetään myöhemmässä vaiheessa poistamaan tietoa aiemmasta solun tilasta $\mathbf{C}^{t-1}$ kertomalla ne elementtikohtaisesti.

Syötteen ja kandidaattiarvon välisinä painokertoimina käytetään $\mathbf{I} \times \mathbf{J}$ matriisia $\mathbf{W_{xz}}$, ja aiemman piilokerroksen ja kandidaattiarvon välisinä painokertoimina $\mathbf{J} \times \mathbf{J}$ matriisia $\mathbf{W_{hz}}$. Määritellään kandidaattiarvo

$$\mathbf{Z}^t = \tanh(\mathbf{X}^t\mathbf{W_{xz}} + \mathbf{H}^{t-1}\mathbf{W_{hz}}). \tag{4.3}$$



Kuva 9: LSTM-verkon piilokerros kohdassa t, josta näkyy unohtamisportista saatu arvo $\mathbf{F}^t$, syöteportista saatu arvo $\mathbf{I}^t$, kandidaattiarvo $\mathbf{Z}^t$ ja tulosportista saatu arvo $\mathbf{O}^t$.

Kandidaattiarvot voidaan nähdä solun tilaan seuraavaksi säilytettävän tiedon ehdotuksena.

Syöteportin avulla päätetään mitä uudesta kandidaattiarvosta säilytetään, ennen kuin se lisätään solun tilaan. Syötteen ja syöteportin välisinä painokertoimina on $\mathtt{I} \times \mathtt{J}$ matriisi $\mathbf{W_{xi}}$, ja aiemman piilokerroksen ja syöteportin välisinä painokertoimina $\mathtt{J} \times \mathtt{J}$ matriisi $\mathbf{W_{hi}}$. Määritellään syöteportista saatu arvo

$$\mathbf{I}^t = \sigma(\mathbf{X}^t\mathbf{W_{xi}} + \mathbf{H}^{t-1}\mathbf{W_{hi}}), \tag{4.4}$$

jossa jälleen käytetään logistista sigmoidia σ sijoittamaan arvot välille $[0, 1]$.

Lasketaan uusi solun tila $\mathbf{C}^t$ unohtamalla turha tieto edellisestä solun tilasta $\mathbf{C}^{t-1}$ ja ottamalla mukaan tärkeä tieto kandidaattiarvosta $\mathbf{Z}^t$

$$\mathbf{C}^t = \mathbf{F}^t \odot \mathbf{C}^{t-1} + \mathbf{I}^t \odot \mathbf{Z}^t. \tag{4.5}$$

Tulosportin arvo saadaan kaavojen (4.1) ja (4.4) tapaan, jossa syötteen ja tulosportin välisinä painokertoimina $\mathbf{I} \times \mathbf{J}$ matriisi $\mathbf{W_{xo}}$, ja aiemman piilokerroksen ja tulosportin välisinä painokertoimina $\mathbf{J} \times \mathbf{J}$ matriisi $\mathbf{W_{ho}}$. Määritellään tulosportista saatu arvo

$$\mathbf{O}^t = \sigma(\mathbf{X}^t\mathbf{W_{xo}} + \mathbf{H}^{t-1}\mathbf{W_{ho}}). \tag{4.6}$$

Tulosportin vastuu on ottaa solun tilan aktivaatiosta $tanh(\mathbf{C}^t)$ tarvittava tieto ja muodostaa näin piilokerroksesta askeleessa t saatu arvo

$$\mathbf{H}^t = \mathbf{O}^t \odot \tanh(\mathbf{C}^t). \tag{4.7}$$

Piilokerroksen arvoa käytetään sen jälkeen selvittämään neuroverkon tulos $\mathbf{Y}^t$ käyttämällä kaavaa (3.2).

4.2 Taaksepäin kulku

Taaksepäin kulkiessa lasketaan jälleen virheet askeleittain loppupäästä alkaen. Tuloskerroksen muutos $\Delta\mathbf{Y}$ on määritelty kaavan (3.3) mukaan. Piilokerroksen muutosta laskiessa otetaan huomioon huomioon tuloskerroksen muutos, sekä painotettuna seuraavan askeleen kandidaattiarvon muutos $\Delta\mathbf{Z}^{t+1}$ ja porteista saatujen arvojen muutokset $\Delta\mathbf{I}^{t+1}$, $\Delta\mathbf{F}^{t+1}$, $\Delta\mathbf{O}^{t+1}$. Ensimmäiseksi laskettavaa eli viimeisen askeleen piilokerroksen muutosta laskettaessa tarvittavat muutokset $\Delta\mathbf{Z}^{T+1}$, $\Delta\mathbf{I}^{T+1}$, $\Delta\mathbf{F}^{T+1}$ ja $\Delta\mathbf{O}^{T+1}$ ovat nollamatriiseja $\mathbf{0_{1\times J}}$, kun sekvenssin pituus on T ja piilokerroksen leveys on J. Piilokerroksen muutos määritellään

$$\Delta\mathbf{H}^t = \Delta\mathbf{Y}^t + \Delta\mathbf{Z}^{t+1}\mathbf{W}_{\mathbf{hz}}^\intercal + \Delta\mathbf{I}^{t+1}\mathbf{W}_{\mathbf{hi}}^\intercal + \Delta\mathbf{F}^{t+1}\mathbf{W}_{\mathbf{hf}}^\intercal + \Delta\mathbf{O}^{t+1}\mathbf{W}_{\mathbf{ho}}^\intercal. \tag{4.8}$$

Solun tilan muutos saadaan piilokerroksen muutosta käyttämällä määriteltyä

$$\Delta\mathbf{C}^t = \Delta\mathbf{H}^t \odot \mathbf{O}^t \odot \tanh'(\mathbf{C}^t). \tag{4.9}$$

Unohtamisportin, syöteportin ja tulosportin muutokset määritellään

$$\begin{aligned} \Delta\mathbf{F}^t &= \Delta\mathbf{C}^t \odot \mathbf{C}^{t-1} \odot \sigma'(\mathbf{F}^t), \\ \Delta\mathbf{I}^t &= \Delta\mathbf{C}^t \odot \mathbf{Z}^t \odot \sigma'(\mathbf{I}^t), \\ \Delta\mathbf{O}^t &= \Delta\mathbf{H}^t \odot \tanh(\mathbf{C}^t) \odot \sigma'(\mathbf{O}^t), \end{aligned} \tag{4.10}$$

jossa logistisen sigmoidin σ derivaatta on

$$\sigma'(x) = \sigma(x)(1 - \sigma(x)). \tag{4.11}$$

Jälleen kandidaattiarvon muutos määritellään

$$\Delta \mathbf{Z}^t = \Delta \mathbf{C}^t \odot \mathbf{I}^t \odot \tanh'(\mathbf{Z}^t). \tag{4.12}$$

Ylempiä määrittelyjä käyttämällä lasketaan piilokerroksen ja tuloskerroksen välisten painokerrointen muutos $\Delta \mathbf{W_y}$ käyttämällä kaavaa (3.5), piilokerrosten askelien väliset painokerrointen muutokset

$$\begin{aligned}
\Delta \mathbf{W_{hz}} &= \sum_t \mathbf{H}^{(t)\intercal} \Delta \mathbf{Z}^{t+1}, \\
\Delta \mathbf{W_{hi}} &= \sum_t \mathbf{H}^{(t)\intercal} \Delta \mathbf{I}^{t+1}, \\
\Delta \mathbf{W_{hf}} &= \sum_t \mathbf{H}^{(t)\intercal} \Delta \mathbf{F}^{t+1}, \\
\Delta \mathbf{W_{ho}} &= \sum_t \mathbf{H}^{(t)\intercal} \Delta \mathbf{O}^{t+1},
\end{aligned} \tag{4.13}$$

sekä syötekerroksen ja piilokerroksen väliset painokerrointen muutokset

$$\begin{aligned}
\Delta \mathbf{W_{xz}} &= \sum_t \mathbf{X}^{(t)\intercal} \Delta \mathbf{Z}^{t}, \\
\Delta \mathbf{W_{xi}} &= \sum_t \mathbf{X}^{(t)\intercal} \Delta \mathbf{I}^{t}, \\
\Delta \mathbf{W_{xf}} &= \sum_t \mathbf{X}^{(t)\intercal} \Delta \mathbf{F}^{t}, \\
\Delta \mathbf{W_{xo}} &= \sum_t \mathbf{X}^{(t)\intercal} \Delta \mathbf{O}^{t}.
\end{aligned} \tag{4.14}$$

Takaisinkytkeytyvien neuroverkkojen tapaan nämä painokerrosten muutokset lisätään painokertoimiin ja jatketaan seuraavaan opetusesimerkkiin.

5 Pohdinta

Tutkielman tarkoituksena oli yksinkertaisten neuroverkkojen toteutus, joiden opettamisessa käytettiin online-menetelmää. Menetelmässä kunkin opetusesimerkin jälkeen painokertoimia siirretään neuroverkon approksimaation tarkentamiseksi. Mahdollista on myös ottaa kerralla useamman opetusesimerkin, niin sanotun pienerän (mini-batch), painokerrointen muutokset ja käyttää niiden keskiarvoa painokerrointen siirtämiseen. Pienerien käyttäminen voi usein olla tehokkaampaa neuroverkkoa opettaessa (Sutskever, 2013). Tutkielmassa myös käsiteltiin vain yhden piilokerroksen sisältäviä neuroverkkoja, mutta tosielämän ongelmiin käytetään usein useampaa piilokerrosta (Lecun et al., 2015).

Tutkielmassa käytetty LSTM on Gers et al. (1999) kehittämä versio, jossa verkkoon on lisätty unohtamisportit. Monissa tapauksissa aikaisemman

tiedon huomioon ottaminen syötettä laskiessa voi jäädä tarpeettomaksi. Esimerkiksi tekstiä lukiessa seuraavaan lauseeseen tai kappaleeseen siirtyessä aikaisemmassa osassa voi olla paljon tietoa josta nykyinen osa ei ole riippuvainen. LSTM:n alkuperäisessä versiossa unohtamisportteja ei ollut, ja tutkijat huomasivat, että solun tilan arvo kasvaa jatkuvasti sekvenssiä prosessoidessa ja tarvitsi mekanismin tarpeettomaksi jääneen tiedon poistamiseen. LSTM:n eri versioita on vertailtu esimerkiksi Greff et al. (2015) toimesta.

LSTM-verkkojen opettamista voidaan edelleen pyrkiä tehostamaan hyödyntämällä muita koneoppimisen tai neuroverkkojen tekniikoita. Konvoluutioverkkoja käytetään usein kuvista tai spektrogrammeista havaittavien piirteiden mallintamiseen. Konvoluutioverkosta saatuja malleja tai representaatioita voidaan sen jälkeen käyttää LSTM-verkon syötteinä (Lecun et al., 2015). Tekstisyötteitä käsitellessä oppimista voidaan tehostaa esittämällä sanat lähisukulaisuudet esittävinä vektoriesityksinä. LSTM voi näin hyödyntää suoraan sanojen synonyymeja, antonyymeja ja muita sukulaisuuksia oppiessa edelleen lauseiden ja pitempien tekstien riippuvaisuuksia. Yksi tällainen sanojen vektoriesityksiin kehitetty tekniikka on GloVe (Pennington et al., 2014).

Viitteet

Gers, F. A., , Schmidhuber, J., and Cummins, F. (1999). Learning to forget: Continual prediction with lstm. *Neural Computation*, 12:2451–2471.

Good, O. (2015). How google translate squeezes deep learning onto a phone. `http://googleresearch.blogspot.fi/2015/07/how-google-translate-squeezes-deep.html`. Accessed: 2016-02-20.

Graves, A. (2012). *Supervised Sequence Labelling with Recurrent Neural Networks*. Springer.

Greff, K., Srivastava, R. K., Koutník, J., Steunebrink, B. R., and Schmidhuber, J. (2015). Lstm: A search space odyssey. *arXiv preprint arXiv:1503.04069*.

Hochreiter, S., Bengio, Y., Frasconi, P., and Schmidhuber, J. (2001). Gradient flow in recurrent nets: the difficulty of learning lon-term dependencies. In *A Field Guide to Dynamical Recurrent Networks*, pages 237–243. IEEE Press.

Hochreiter, S. and Schmidhuber, J. (1995). Long short-term memory. *Neural Computation*, 9:1735–1780.

Lecun, Y., Bengio, Y., and Hinton, G. (2015). Deep learning. *Nature*, 521:436–444.

Pennington, J., Socher, R., and Manning, C. D. (2014). Glove: Global vectors for word representation. In *Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543.

Sutskever, I. (2013). *Training recurrent neural networks*. PhD thesis, University of Toronto.

Neuroverkkojen käyttö go-pelissä

Toni Leino

Tiivistelmä

Go on jo vuosikymmenien ajan ollut vaikea peli tietokoneohjelmille. Vaikka go-ohjelmat ovatkin viime vuosina kehittyneet valtavasti erityisesti Monte Carlo -puuhaun ansiosta, go on edelleenkin yksi vaikeimmista lautapeleistä. Aivan lähiaikoina on kuitenkin saavutettu hyviä tuloksia neuroverkkoja käyttämällä ja Googlen kehittämä AlphaGo on ensimmäinen go-ohjelma, joka on voittanut ammattilaispelaajan.

Avainsanat: go, neuroverkot, syväoppiminen, tekoäly

1 Johdanto

Go on tuhansia vuosia vanha aasialainen lautapeli. Vaikka peli on säännöiltään erittäin yksinkertainen, se on tietokoneille huomattavasti vaikeampi kuin esimerkiksi shakki ja sitä on jo kauan pidetty yhtenä suurimpina koneoppimisen haasteista. [4]

Neuroverkko on yleinen koneoppimismenetelmä, joka muistuttaa ihmisaivojen toimintaa. Neuroverkkoja on käytetty monissa peleissä, mutta yleensä niillä ei saavuteta yhtä hyviä tuloksia kuin muilla algoritmeilla, jotka perustuvat "raakaan voimaan" (brute force) [8].

Algoritmit, jotka toimivat muissa peleissä hyvin, eivät kuitenkaan usein pärjää gossa. Tässä tutkielmassa selvitän, mitä sellaisia ongelmia gossa on, joista perinteiset ohjelmat eivät selviä, mutta jotka saattavat olla mahdollisia ratkaista neuroverkkoja käyttämällä.

Toinen luku kertoo gon säännöt ja esittelee joitakin algoritmeja, joita gota ja muita pelejä pelaavat ohjelmat yleensä käyttävät. Tärkeää tässä luvussa ovat erityisesti algoritmien ongelmat.

Kolmas luku käsittelee neuroverkkoja yleisellä tasolla: miten neuroverkot toimivat, miten niitä opetetaan ja mihin niitä voidaan käyttää. Tässä luvussa keskitytään lähinnä sellaisiin neuroverkkoihin, joita on käytetty go-ohjelmissa.

Neljännessä luvussa esitellään neuroverkkojen käyttöä gossa: mitä algoritmeja ne käyttävät, mitä syötteitä niille annetaan, miten niitä opetetaan ja mitä hyötyjä niissä on muihin ohjelmiin verrattuna.

Viidennessä luvussa on lyhyt yhteenveto.

2 Taustaa

2.1 Go-peli

2.1.1 Yleistä

Golla on paljon yhteistä esimerkiksi shakin kanssa: siinä on kaksi pelaajaa, se on deterministinen (sattuma ei vaikuta lopputulokseen) ja se on täydellisen tiedon peli, eli molemmat pelaajat tietävät koko pelilaudan tilan (toisin kuin esimerkiksi monissa korttipeleissä, joissa pelaaja ei näe muiden pelaajien kortteja) [9].

Shakkiin verrattuna go on kuitenkin erittäin vaikea peli tietokoneille. Tämä johtuu muun muassa pelilaudan suuresta koosta (yleisimmin käytetty pelilauta on yli viisi kertaa isompi kuin shakin pelilauta), pelin kestosta ja siitä, että yksittäisen kiven (gon pelinappulan) vaikutusta peliin on vaikea arvioida. Nämä kaikki vaikuttavat toisiinsa: esimerkiksi jokin siirto saattaa vaikuttaa peliin merkittävästi vasta kymmeniä tai jopa satoja vuoroja myöhemmin, mikä on mahdollista pelin pituuden takia.

2.1.2 Gon säännöt

Gossa, kuten monessa muussakin lautapelissä, käytetään monenlaisia sääntöjä maasta ja pelaajista riippuen. Tämän luvun tavoitteena on selittää yleisesti, mistä pelissä on kyse ja miten säännöt tulee ottaa huomioon go-ohjelmissa. Tarkemmat säännöt (ja lisäksi eri maiden välisiä eroja) löytyvät esimerkiksi Wikipediasta [15].

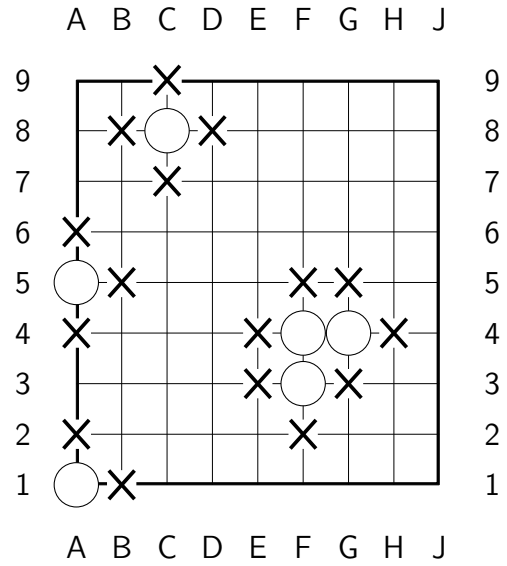
Pelaajat (musta ja valkoinen) asettavat vuorotellen nappuloitaan (“kiviä”) pelilaudan viivojen risteyskohtiin. Musta pelaaja aloittaa. Pelilauta on yleensä 19x19-kokoa, mutta myös esimerkiksi 9x9- ja 13x13-lautoja käytetään.

Tavoitteena on ympäröidä mahdollisimman suuri alue laudasta omilla kivillä. Vastustajan kiven voi vangita asettamalla oman kiven kaikkiin sitä ympäröiviin risteyksiin. Jos kiviä on monta kiinni toisissaan (“ketju”), ne täytyy kaikki vangita kerralla. Omaa kiveä ei saa asettaa sellaiseen paikkaan, jossa se tulisi heti vangituksi, paitsi jos onnistuu samalla vangitsemaan vastustajan kiviä.

Kuvassa 1 on esimerkkejä kivien vangitsemisesta. Musta pelaaja voi vangita vasemmalla olevat valkoiset kivet asettamalla oman kivensä X:llä merkittyihin kohtiin. Oikealla olevat kivet muodostavat ketjun, joten niiden vangitsemiseen tarvitaan enemmän kiviä.

“Itsemurhan” lisäksi kiellettyjä ovat myös sellaiset siirrot, jotka voisivat johtaa loputtomaan peliin. Tähän liittyvät säännöt vaihtelevat maittain, mutta yleisesti ottaen pelaaja ei saa pelata siirtoa, jonka jälkeen päädyttäisiin johonkin aiempaan pelitilanteeseen.

Kiven sanotaan olevan *elävä*, jos vastustaja ei pysty vangitsemaan sitä. Muussa tapauksessa kivi on *kuollut*. Ketju on elävä, jos sen sisällä on vähin-



Kuva 1: Esimerkki kivien vangitsemisesta.

tään kaksi erillistä aluetta (“silmää”). Kuvassa 2 on esimerkkejä tästä. Kivien vangitsemiseksi vastustajan tulisi asettaa oma kivensä sekä ruksilla että kolmiolla merkittyyn kohtaan, mutta itsemurhasäännön takia kumpikaan siirto ei ole sallittu.

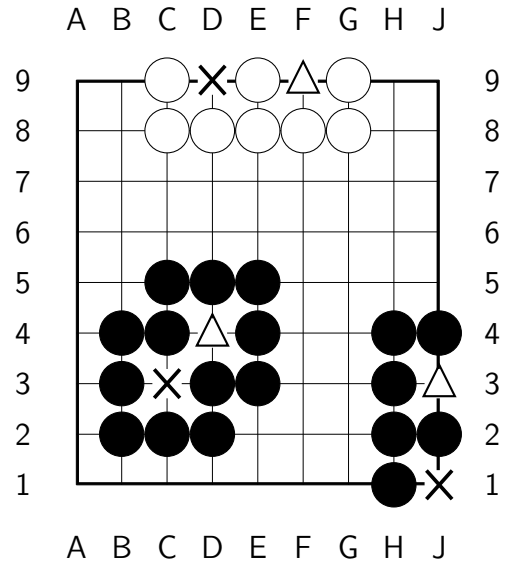
Gossa ei ole tarkkaan määritelty, milloin peli päättyy. Sen sijaan pelaaja voi halutessaan ohittaa vuoronsa ja peli on ohi, kun molemmat pelaajat ohittavat vuoronsa peräkkäin. Käytännössä tämä tapahtuu silloin, kun kaikki mahdolliset alueet on jo vallattu eikä ole enää olemassa siirtoja, joilla saisi lisää pisteitä.

2.1.3 Pelin voittaminen

Gossa pelin voittaa se pelaaja, joka saa enemmän pisteitä. Pistelaskutapoja on useita erilaisia, mutta yleensä voittaja on sama riippumatta siitä, mitä menetelmää käytetään. Tämän vuoksi esittelen vain yhden niistä, japanilaisen menetelmän. [10]

Japanilaisessa pistelaskutavassa pisteitä saa pelaajan valtaamasta alueesta (yksi piste jokaisesta ympäröidystä viivojen risteyskohdasta) ja vangituista vastustajan kivistä (mukaan lukien kuolleet kivet, koska ne tulisivat vangituksi, jos peliä jatkettaisiin; jokainen kivi on yhden pisteen arvoinen). Alue on vallattu silloin, kun vastustaja ei voi asettaa eläviä kiviä sinne; alueen ei ole pakko olla kokonaan ympäröity pelaajan omilla kivillä. Lisäksi valkoinen pelaaja saa tasoitusta (*komi*) 6.5 pistettä, sillä mustalla pelaajalla on aloittajan etu.

Pistelaskutavasta riippumatta eniten pisteitä kerännyt pelaaja voittaa.



Kuva 2: Esimerkkejä ketjuista, joilla on kaksi silmää ja joita ei siis voi vangita.

Pistemäärien erolla ei ole merkitystä.

2.1.4 Pelaajien taitotason arviointi

Gossa on kaksi yleisesti käytettyä tapaa arvioida pelaajien taitotasoa: Elo-luku ja *kyu*- ja *dan*-tasot.

Elo-luku on Arpad Elon 1960-luvulla kehittämä menetelmä. Elo-lukua käytettiin alun perin shakissa, mutta nykyään sitä käytetään monessa muussakin pelissä. Pelaajan Elo-luku on yleensä aluksi 1000 ja laskee tai nousee pelitulosten perusteella. Pelin jälkeen lasketaan pelaajien Elo-lukujen perusteella pelituloksen odotusarvo ja verrataan sitä pelin oikeaan tulokseen. Tämän jälkeen häviäjän Elo-luvusta siirretään pisteitä voittajalle. Mitä suurempi odotusarvon ja todellisen tuloksen erotus on, sen enemmän pisteitä siirretään. Esimerkiksi 200 pisteen ero Elo-luvuissa tarkoittaa, että paremman pelaajan pitäisi voittaa noin 75 % peleistä.

Kyu- ja dan-tasot ovat perinteinen tapa merkitä go-pelaajien vahvuuksia. Kyu-tasot (lyhennetään k) ovat aloittelijoille ja keskitason pelaajille ja dan-tasot (lyhennetään d) edistyneemmille pelaajille. Pelaajan taso voi olla esimerkiksi 10k tai 5d. Kyu-tasoilla pienempi luku on parempi, mutta dan-tasoilla on toisinpäin. Esimerkiksi $5\text{k} < 1\text{k} < 1\text{d} < 5\text{d}$. Juuri säännöt oppineen pelaajan taso on noin 30k ja paras taso on yleensä noin 9d. Ammattilaispelaajille on omat dan-tasonsa, joita merkitään p-kirjaimella, mutta ne perustuvat enemmänkin pelikokemukseen kuin taitoon.

Toisin kuin Elo-luku, kyu- tai dan-tason määrittämiseen ei ole olemassa tarkkaa kaavaa. Käytännössä se on arvio siitä, paljonko tasoitusta huonompi

pelaaja tarvitsee, jotta hänellä olisi 50 % mahdollisuus voittaa. Jos esimerkiksi pelaajan A taso on 10k ja B:n taso on 15k, B saa asettaa pelin alussa viisi ylimääräistä kiveä laudalle.

Kummallekin menetelmälle yhteistä on, että ne kertovat ainoastaan pelaajan tason suhteessa muihin pelaajiin. Tämä tarkoittaa, että erimaalaisten pelaajien tulokset eivät vertailukelpoisia keskenään, sillä pelaajien taso vaihtelee eri maissa. Lisäksi pelaajien keskitason noustessa kaikkien pelaajien luokitus laskee ja päinvastoin.

2.1.5 Go-ohjelmien haasteita

Vaikka säännöt ovatkin yksinkertaiset, ne asettavat muutamia haasteita go-ohjelmille. Esimerkiksi pelilaudan suuren koon takia on vaikeaa tutkia kaikkia mahdollisia siirtoja, ja lisäksi siirtojen vaikutusta voi olla vaikeaa arvioida. Käytännössä pelilaudan voi ajatella koostuvan useista pienistä pelilaudoista, jotka kaikki vaikuttavat toisiinsa [1]. Tämän vuoksi osa go-ohjelmista keskittyykin ainoastaan paikallisiin siirtoihin, eli ne tutkivat ainoastaan viimeksi pelatun kiven ympäristöä [14].

Toinen ongelma on, että gossa ei ole selkeästi määritelty, milloin peli päättyy. Tämä tarkoittaa, että kokonaisia pelejä pelaavan go-ohjelman (toisin kuin yksittäisiä siirtoja ennustavan) täytyy pystyä tunnistamaan laudalta elävät ja kuolleet kivet ja tämän avulla päättelemään, milloin peli kannattaa lopettaa [1].

2.2 Go-ohjelmien toiminta

2.2.1 Pelipuu

Pelipuu (game tree) on rakenne, joka esittää esimerkiksi go- tai shakkipelin kulun. Puun solmut ovat mahdollisia pelitilanteita ja solmun lapsisolmuja ovat ne tilanteet, jotka voidaan saavuttaa seuraavalla siirrolla. Puun juurisolmu on pelin alkutilanne ja lehdet ovat tilanteita, joissa peli on päättynyt.

2.2.2 Minimax-algoritmi

Minimax on jo 1920-luvulla kehitetty algoritmi, jolla voidaan etsiä pelipuusta paras mahdollinen siirto. Vaikka minimax onkin yksinkertainen, se on vielä nykyäänkin erittäin yleisesti käytetty algoritmi: muun muassa lähes kaikki shakkiohjelmat käyttävät sitä. [8]

Minimax toimii seuraavasti: oletetaan, että pelaajat ovat A ja B. Puun solmut (eli pelitilanteet) voidaan pisteyttää niin, että A:n voitto on 1 piste, B:n voitto on -1 piste ja tasapeli on 0 pistettä. Tällöin A:n kannattaa valita aina se siirto, joka johtaa suurimpaan pistemäärään ja B:n tulisi valita pienin pistemäärä.

Sisäsolmut, eli tilanteet, joissa peli on vielä kesken, pisteytetään niin, että valitaan lapsisolmuista paras (pienin tai suurin pistemäärä riippuen siitä, kumman pelaajan vuoro on). Jos nekin ovat sisäsolmuja, toistetaan samaa menetelmää rekursiivisesti kunnes päädytään lehtisolmuun.

Käytännössä pelipuu on aivan yksinkertaisimpia pelejä lukuun ottamatta niin suuri, että sitä ei voi tutkia kokonaan. Esimerkiksi gossa on arvioitu olevan noin 10^{170} mahdollista tilannetta [13]. Tämän vuoksi minimax-haku keskeytetään, kun edetään puussa liian syvälle ja arvioidaan pelitilanne *arviointifunktion* (evaluation function) avulla. Arviointifunktio palauttaa arvon välillä $[-1, 1]$.

Minimax-algoritmia voi nopeuttaa *alpha-beta-karsimisella* (alpha-beta pruning). Se on algoritmi, joka poistaa pelipuusta sellaisia siirtoja, jotka ovat varmasti huonompia kuin jokin aiemmin löydetty siirto [5]. Alpha-beta -karsiminen ei vaikuta valittuun siirtoon, mutta sen avulla kyseinen siirto löytyy yleensä nopeammin.

2.2.3 Monte Carlo -puuhaku

Minimax-algoritmi ei toimi gossa niin hyvin kuin esimerkiksi shakissa. Yksi syy tähän on, että gohon on huomattavasti vaikeampaa kehittää hyvä arviointifunktio [1]. Viime vuosina parhaat go-ohjelmat ovat käyttäneet minimaxin sijaan Monte Carlo -hakua, joka ei tarvitse arviointifunktiota [3].

Monte Carlo -puuhaku simuloi satunnaisia pelejä ja valitsee sen siirron, jolla saavutettiin parhaat tulokset näissä peleissä. Tällä algoritmilla saavutetaan erittäin hyviä tuloksia gossa, vaikka se ei vaadi mitään tietoa pelistä, toisin kuin Minimax-algoritmin arviointifunktio [3].

3 Neuroverkoista

3.1 Yleistä

Tämä luku perustuu enimmäkseen Haykinin Neural Networks -kirjaan [6].

Neuroverkko (neural network) on koneoppimismenetelmä, joka muistuttaa aivojen toimintaa. Neuroverkko rakentuu *neuroneista*, joka toimii kuin yksittäinen hermosolu.

Neuroverkot toimivat hyvin tilanteissa, joissa ei ole selkeitä sääntöjä. Niitä käytetään muun muassa kuvantunnistukseen ja automaattiseen kääntämiseen (esimerkiksi Google Translate).

Jotta neuroverkolla voidaan tehdä mitään hyödyllistä, sitä täytyy opettaa. Opettaminen voi tapahtua esimerkiksi siten, että annetaan neuroverkolle ongelmia, joiden ratkaisut tiedetään. Verkon ratkaisuja verrataan oikeisiin ratkaisuihin ja tämän jälkeen tehdään muutoksia verkkoon, jotka parantavat tulosta.

3.2 Neuroverkon toiminta

Neuroni saa yhden tai useampia syötearvoja, joiden oletetaan tässä olevan lukuja välillä $[-1, 1]$. Jokainen syötearvo kerrotaan painokertoimella (jokaista syötettä varten on oma painokertoimella), jonka jälkeen ne summataan yhteen ja lopuksi *aktivaatiofunktio* muuttaa tämän summan arvoksi välillä $[-1, 1]$.

Neuroverkossa on ensimmäisenä syötekerros, joka sisältää neuroverkolle annetut syötearvot. Tämän lisäksi neuroverkossa on neuroneista koostuva tuloskerros ja näiden välissä yksi tai useampia piilotettuja kerroksia. Syötekerroksen arvot annetaan syötteeksi ensimmäisen piilokerroksen neuroneille, niiden tulokset syötetään eteenpäin seuraavaan kerrokseen ja prosessia toistetaan, kunnes päädytään tuloskerrokseen.

Neuroverkon syötteiden ja tuloksien lukumäärä voi vaihdella käyttötarkoituksesta riippuen. Esimerkiksi kuvantunnistusongelmassa syötteitä voivat olla kuvan pikselit ja tulosneuroneita voi olla yksi jokaista mahdollista kuvatyyppiä varten [7].

3.3 Neuroverkon opettaminen

Eräs neuroverkkojen eduista esimerkiksi Monte Carlo -puuhakuun verrattuna on, että neuroverkot kykenevät oppimaan uusia asioita. Neuroverkon opettaminen tapahtuu yleensä muuttamalla painokertoimia, mutta myös neuronien lisääminen tai poistaminen tai niiden välisten yhteyksien muuttaminen on mahdollista.

Oppimismenetelmät voidaan jakaa kolmeen ryhmään: ohjattu oppiminen, ohjaamaton oppiminen ja vahvistusoppiminen. *Ohjatussa oppimisessa* (supervised learning) neuroverkon tehtävä on luokitella sinne annetut syötteet. Tämän voi toteuttaa esimerkiksi niin, että neuroverkon viimeisessä kerroksessa on yksi neuroni jokaista mahdollista luokkaa varten ja näistä valitaan se, jonka tulosarvo on suurin.

Tyypillinen esimerkki ohjatusta oppimisesta on erilaiset kuvantunnistusongelmat. Esimerkiksi käsinkirjoitetun tekstin tunnistuksessa ensin erotellaan kirjaimet (tai muut kirjoitusmerkit) toisistaan ja tämän jälkeen tunnistetaan ne neuroverkon avulla.

Ohjaamaton oppiminen (unsupervised learning) on muuten samanlaista kuin ohjattu oppiminen, mutta siinä ei ole ennalta määritettyjä luokkia. Neuroverkon täytyy siis jakaa syötteet ryhmiin, joissa ryhmän alkiot ovat samanlaisia keskenään, mutta erilaisia kuin muiden ryhmien alkiot.

Vahvistusoppimisessa (reinforcement learning) neuroverkko saa syötteeksi *ympäristön* (environment) tilan ja valitsee sen perusteella toiminnon. Tavoitteena on valita mahdollisimman hyvää palautetta tuottava toiminto. Vahvistusoppiminen on yleisin peleissä käytetty menetelmä, ja sitä voidaan käyttää myös esimerkiksi roboteissa.

3.4 Syväoppiminen

Yksinkertainen neuroverkko ei sovellu kovin hyvin monimutkaisten ongelmien ratkaisemiseen, sillä ratkaisuun tarvittava neuroverkko olisi erittäin suuri ja lisäksi ratkaisua on vaikea yleistää. Esimerkiksi kuvantunnistusongelmat voivat olla haastavia. Ratkaisuna tähän on syväoppiminen, jossa verkossa on useita kerroksia, jotka käsittelevät ongelmaa eri tasoilla. Kuvia tunnistavassa verkossa yksi kerros voi etsiä kulmat ja reunat, toinen niiden muodostamat kuviot ja kolmas tutkii kuvioiden sijaintia. Tämän arvellaan vastaavan ihmisen näköjärjestelmän toimintaa [8].

Syväoppivilla neuroverkoilla on saavutettu erittäin hyviä tuloksia monenlaisissa ongelmissa. Shakissa Giraffe-ohjelma on lähellä parhaita ihmispelaajia, mutta häviää silti selvästi muille shakkiohjelmille [8]. Kuvantunnistuksessa syväoppivalla neuroverkolla saavutettiin myös huomattavasti parempia tuloksia kuin yksinkertaisemmilla verkoilla [7].

4 Neuroverkot ja go

4.1 Yleistä

Neuroverkkopohjaisia go-ohjelmia on ollut olemassa jo vuosikymmeniä, mutta viime aikoihin asti ne ovat olleet huonompia pelaajia kuin puuhakuun perustuvat ohjelmat. Viime vuosina ne ovat kuitenkin kehittyneet huomattavasti muun muassa syväoppimisen ansiosta.

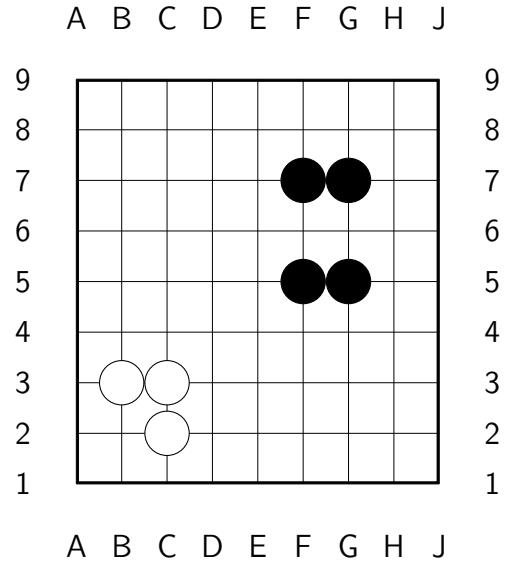
Neuroverkon tavoitteena ei välttämättä ole etsiä parasta siirtoa, vaan niitä voi käyttää myös esimerkiksi arviointifunktiona puuhakualgoritmeissa [12]. Vielä erikoisempi käyttökohde olisi päättää, milloin kannattaa käyttää enemmän aikaa siirron valintaan. Gossa, samoin kuin shakissakin, pelaajalla on vain rajallisesti aikaa tehdä siirtonsa eikä kelloa nollata siirtojen välissä (säännöt kuitenkin vaihtelevat). Lai [8] arveleekin, että neuroverkkoa voisi käyttää ajanhallintaan.

Go-ohjelmissa käytettyjä neuroverkkoja opetetaan eri tavoin, mutta yleisimmät ovat ohjattu oppiminen käyttäen ammattilaispelaajien pelejä [2] ja vahvistusoppiminen, jossa neuroverkko pelaa joko itseään tai muita ohjelmia vastaan [12].

4.2 Go hahmontunnistusongelmana

Gon strategiat perustuvat usein *muotoon* (shape). Tämä tarkoittaa, että pelaajat etsivät laudalta kivien muodostamia kuvioita ja käyttävät näitä heuristiikkoina [1] [11]. Kuvassa 3 on esimerkkejä huonosta ja hyvästä muodosta. Valkoisten kivien muodostama “tyhjä kolmio” on huono muoto, josta ei ole pelaajalle juuri mitään hyötyä. Sen sijaan mustien kivien muodostama

"bambuliitos" on hyvä muoto, koska vastustaja ei pysty erottamaan kiviä toisistaan, vaan ne voi aina yhdistää ketjuksi.



Kuva 3: Esimerkki hyvästä ja huonosta muodosta.

Clark ja Storkey [2] arvioivatkin, että hahmontunnistusta hyödyntämällä voidaan kehittää huomattavasti tehokkaampia go-ohjelmia. Sopivalla hahmontunnistusalgoritmilla voidaan karsia pois suuri osa siirroista, jolloin loput siirrot saadaan tutkittua tehokkaammin. Clarkin ja Storkeyn kehittämä neuroverkkopohjainen ohjelma pelaa huonommin kuin parhaat go-ohjelmat, mutta se käyttää selvästi vähemmän aikaa siirron valitsemiseen [2].

4.3 DeepMind ja AlphaGo

DeepMind on Googlen omistama yritys, joka on kehittänyt muun muassa erilaisia pelejä pelaavia neuroverkkoja. DeepMindin itseoppivat ohjelmat ovat jopa ihmispelaajia parempia [4].

DeepMindin kehittämä AlphaGo on ensimmäinen ja tähän mennessä ainoa ohjelma, joka on voittanut mestaritason ihmispelaajan [12]. AlphaGo käyttää useita syväoppivia neuroverkkoja, jotka sekä etsivät parhaita siirtoja että arvioivat pelitilanteita. Verkot oppivat ohjatusti mestarien pelaamista peleistä, mutta myös vahvistusoppimista käytetään antamalla verkkojen pelata itseään vastaan.

AlphaGon parhaat tulokset saavutetaan yhdistämällä neuroverkot ja Monte Carlo -puuhaku. Tällä algoritmilla AlphaGo voittaa yli 99 % peleistä muita go-ohjelmia vastaan [12].

4.4 Go-ohjelmien tuloksia

Vaikka go-ohjelmia on ollut jo 1960-luvulta lähtien, ne ovat viime aikoihin asti olleet selvästi ihmispelaajia huonompia. Shakissa Deep Blue -ohjelma voitti maailmanmestarin jo vuonna 1997, mutta yksi sen ajan parhaista go-ohjelmista, Many Faces of Go, oli tasoltaan vain noin 3 kyu, joka vastaa keskitason harrastajapelaajaa [1].

Uudemmista ohjelmista ilmainen GnuGo on saavuttanut KGS-pelipalvelimella tason 5k, mutta se ei käytä kehittyneimpiä algoritmeja. Parhaat kaupalliset ohjelmat, Crazy Stone ja Zen, jotka molemmat perustuvat Monte Carlo -puuhakualgoritmiin, ovat tasoltaan noin 5 dan, eli ne eivät voita parhaimpia harrastajapelaajia. Sen sijaan syväoppiviin neuroverkkoihin perustuvan AlphaGon arvellaan olevan 2–3 dan ammattilaispelaajan tasolla [12].

4.5 Neuroverkkojen etuja ja ongelmia

Yksi merkittävä hyöty neuroverkoissa on niiden oppimiskyky. Toisin kuin esimerkiksi puuhakualgoritmit, ne kykenevät mukautumaan erilaisiin tilanteisiin. Samaa neuroverkkoa pystyy (mahdollisesti pienien muutoksien jälkeen) käyttämään jopa täysin toisessa pelissä [8, s. 33].

Oppimiskyvystä on paljon hyötyä, sillä sen ansiosta neuroverkko pystyy mukautumaan erilaisiin tilanteisiin, mutta siitä voi aiheutua myös ongelmia. Jos neuroverkon opettamiseen käytetyt esimerkit ovat liian samankaltaisia keskenään, voi tapahtua *ylioppimista* (overlearning). Tällöin verkko ei pärjää kovin hyvin tilanteissa, jotka eivät vastaa esimerkkejä.

Neuroverkkopohjainen ohjelma voi olla täysin itseoppiva, eli se ei tiedä aluksi pelistä (tai oikeastaan pelin strategiasta) mitään, tai siinä voi olla jotain tietoa sisäänrakennettuna. Jälkimmäisellä menetelmällä saavutetaan hyviä tuloksia helpommin, sillä tällöin neuroverkon opettamiseen tarvitaan vähemmän esimerkkejä. Gossa tällaista tietoa voi olla esimerkiksi se, että pelitilanne ei käytännössä muutu, jos pelilauta peilataan y-akselin suhteen (tämä pitää yleensä paikkansa myös kuvantunnistuksessa) [2].

Ongelmana sisäänrakennetussa tiedossa on, että siinä voi olla virheitä tai puutteita. Sen sijaan täysin itseoppivalla ohjelmalla on mahdollista löytää jopa täysin uusia strategioita, joita ihmispelaajat eivät ole vielä keksineet [8].

Vaikka neuroverkon opettamiseen voi mennä paljon aikaa, opetetun neuroverkon käyttäminen on nopeaa ja se tutkii huomattavasti pienemmän määrän siirtoja kuin puuhakualgoritmit [8]. Tämä on erityisen hyödyllistä gon kaltaisessa pelissä, jossa mahdollisia siirtoja on valtava määrä.

5 Yhteenveto ja pohdintaa

5.1 Yhteenveto

Gota on jo vuosikymmeniä pidetty erittäin vaikeana pelinä tietokoneille. Puuhakumenetelmillä tietokoneet eivät vieläkään ole parhaimpien ihmispelaajien tasolla, sillä go on liian monimutkainen peli, jotta sen voisi ratkaista raa'an voiman menetelmillä nykypäivän tietokoneilla. Neuroverkoilla, jotka muistuttavat enemmän ihmisaivojen toimintaa, sen sijaan saavutetaan lupaavia tuloksia.

AlphaGo on osoittanut, että go ei olekaan täysin mahdoton ongelma. Tämän tutkimuksen tekohetkellä AlphaGo ei kuitenkaan ole vielä pelannut maailmanmestaria vastaan. Nähtäväksi jää, ovatko tulevaisuuden go-ohjelmat selvästi ihmisiä parempia, kuten shakkiohjelmat ovat olleet jo yli vuosikymmenen ajan.

5.2 Ohjelmien vertailu

Pelien tekoälyjä käsittelevissä tutkimuksissa vertaillaan usein niiden vahvuutta, mutta tähän liittyy eräs ongelma: miten paremmuus määritellään? Sekä Elo-luku että kyu- ja dan-tasot perustuvat voittoprosenttiin: pelaaja A on arvioitu pelaajaa B paremmaksi, jos ja vain jos A:n uskotaan voittavan yli 50% peleistä B:tä vastaan (kun tasapelejä ei lasketa). Koska pelaajan vahvuutta merkitään vain yhdellä luvulla, tästä seuraa, että paremmuus olisi transitiivinen ominaisuus, ts. jos A on B:tä parempi ja B on C:tä parempi, niin A on parempi kuin C. Näin ei kuitenkaan välttämättä ole, sillä esimerkiksi vanhemmissa ohjelmissa oli usein heikkouksia, joita hyödyntämällä ne pystyi helposti voittamaan [1]. On siis mahdollista, että tietokoneohjelma on maailmanmestaria parempi, mutta joku heikommaksi arvioitu pelaaja pystyy silti voittamaan ohjelman hyödyntämällä sen heikkouksia.

5.3 Tekoälyjen tulevaisuudesta

Ovatko neuroverkot parempia gossa kuin puuhakualgoritmit? Ei välttämättä, sillä minimax-algoritmi löytää aina parhaan mahdollisen siirron, jos joko tutkitaan koko pelipuu tai arviointifunktio osaa arvioida tilanteen täydellisesti. Ensimmäinen vaihtoehto tuskin tulee koskaan olemaan mahdollinen, sillä gon pelipuu on liian suuri, mutta on hyvinkin mahdollista, että gohon kehitetään tulevaisuudessa hyvin toimiva arviointifunktio. Se voi kuitenkin olla neuroverkkopohjainen, kuten AlphaGossa.

Vaikka go olisikin ratkaistu ongelma, uusia tekoälyongelmia riittää jatkossakin. DeepMind on esimerkiksi kehittänyt yksinkertaisia videopelejä pelaavan neuroverkon. Merkittävää tässä on se, että sama neuroverkko oppii pelaamaan useita erilaisia pelejä täysin automaattisesti [4]. Tulevaisuuden

tekoälyt voivat siis selviytyä monenlaisista ongelmista ilman ihmisen ohjausta.

Viitteet

[1] B. Bouzy and T. Cazenave, Computer Go: an AI oriented survey. 2001. *Artificial Intelligence* 132, 1, 39–103.

[2] C. Clark and A. Storkey, Training deep convolutional neural networks to play go. Proceedings of the 32nd International Conference on Machine Learning (ICML-15), 2015, 1766–1774.

[3] R. Coulom, Efficient selectivity and backup operators in Monte-Carlo tree search. *Computers and games*. Springer, 2007, 72–83.

[4] DeepMind, DQN. `http://deepmind.com/dqn.html`. Checked 16.2.2016.

[5] P. Doyle, Search Methods. `http://www.cs.duke.edu/brd/Teaching/Previous/AI/Lectures/Summaries/search.html`. Checked 4.1.2016.

[6] S. Haykin, *Neural Networks: A Comprehensive Foundation*. Prentice Hall, 2004.

[7] A. Krizhevsky, I. Sutskever and G. E. Hinton, ImageNet Classification with Deep Convolutional Neural Networks. *Advances in Neural Information Processing Systems 25*, 2012. 1097–1105.

[8] M. Lai, Using Deep Reinforcement Learning to Play Chess. 2015. arXiv:1509.01549.

[9] N. N. Schraudolph, P. Dayan and T. J. Sejnowski, Temporal difference learning of position evaluation in the game of Go. *Advances in Neural Information Processing Systems 6*, 1994, 817–824.

[10] Sensei's Library, Scoring. `http://senseis.xmp.net/?Scoring`. Checked 17.2.2016.

[11] Sensei's Library, Shape. `http://senseis.xmp.net/?Shape`. Checked 1.2.2016.

[12] D. Silver et al., Mastering the game of Go with deep neural networks and tree search. 2016. *Nature* 529, 7587, 484–489.

[13] J. Tromp and G. Farnebäck, Combinatorics of go. *Computers and Games*. Springer, 2007, 84–99.

[14] Erik van der Werf et al., Local Move Prediction in Go. *Computers and Games*. Springer, 2002, 393–412.

[15] Wikipedia, Rules of Go. `https://en.wikipedia.org/wiki/Rules_of_go`. Checked 22.2.2016.

Katsaus keskusyksiköiden energiatehokkuuden kehitykseen

Aleksi Luukko

Tiivistelmä.
Perinteisiä keskusyksiköitä on edelleen lähes joka paikassa, ja ne kuluttavat paljon energiaa ja aiheuttavat muitakin ympäristöongelmia. Keskusyksiköiden energiatehokkuus on parantunut uusien komponenttien myötä, mutta on olemassa myös muita keinoja, joilla niiden energiatehokkuutta voidaan parantaa ja muita ympäristövaikutuksia vähentää. Tässä tutkielmassa käydään läpi, miksi keskusyksiköiden energiatehokkuus ja ympäristövaikutusten hillintä on tärkeää, mitä komponenttivalmistajat ovat tehneet asian hyväksi sekä mitä muita energiatehokkuutta parantavia ja ympäristövaikutuksia pienentäviä keinoja on käytettävissä.

Avainsanat ja -sanonnat: energiatehokkuus, vihreä laskenta, vihreä tietotekniikka, virtualisointi, etärenderöinti, ympäristövaikutukset

1. Johdanto

Maapallon ilmasto on suuressa muutoksessa, ihmisen toiminnan tuloksena. Ilmastonmuutos on todellinen ongelma ja uhka ihmiskunnalle. Kaikki voitava huonon kehityksen katkaisemiseksi on tehtävä. Tässä tietotekniikalla on oma merkityksensä: sen avulla vanhoja paljon ympäristöä kuormittavia järjestelmiä on voitu muuttaa ja muokata digitaalisiksi palveluiksi [Zysman and Huberty 2012]. Tietotekniikan valtava määrä on toisaalta aiheuttanut uusia ympäristöongelmia [Trimi and Park 2012].

Keskusyksiköiden osalta energiatehokkuus on parantunut merkittävästi suhteessa laskentatehoon viimeksi kuluneen vuosikymmenen aikana. Komponenttivalmistajat ovat kehittäneet ja ottaneet käyttöön uusia energiaa säästäviä tekniikoita ja tuoneet markkinoille tehokkaita, mutta energiaa säästäviä tuotteita. Tämän on mahdollistanut laskentatehon tarpeen hidas kasvu varsinkin toimisto- ja hyötykäyttöohjelmistoissa. Neljän vuoden ikäinen keskiverto keskusyksikkö on tarpeeksi tehokas suoriutuakseen näistä vaivatta. Onkin siis aiheellista kiinnittää vielä aikaisempaa enemmän huomiota uusien keskusyksiköiden energiatehokkuuden parantamiseen entisestään ottaen samalla huomioon niiden välilliset ympäristövaikutukset [Trimi and Park 2012].

Käyttämällä useiden käyttäjien kesken jaettuja resursseja voidaan saavuttaa huomattavia säästöjä energiankulutuksessa [Berl and Meer 2010]. Tämän seurauksena yksittäiseltä keskusyksiköltä vaaditun laskentatehon määrä laskee ja voidaan käyttää entistä energiatehokkaampia ja tehottomampia komponentteja.

Jaettujen resurssien käyttöön perustuvat tekniikat ovat yleistyneet vähitellen 2000-luvun aikana [Miller and Pegah 2007].

Tutkielman tavoitteena on muodostaa kokonaiskuva keskusyksiköiden energiatehokkuuden kehityksestä, vaikuttamatta laskentatehoon sekä taaten pienet ympäristövaikutukset komponenttien valmistuksessa ja käytössä. Tutkielmassa käytetään lähinnä keskusyksiköiden *prosessoreja* (CPU) havainnollistamaan energiatehokkuuden kehitystä. Tutkielmassa esitellään myös, miten virtualisoinnin ja etärenderöinnin tekniikoilla voitaisiin vähentää tarvetta erillisille keskusyksiköille säästäen ympäristöä ja varmistaen energiatehokkuuden sekä tarvittavan laskentatehon säilymisen keskitetyissä laskentayksiköissä.

Tutkielman alussa luvussa 2 käsitellään energiatehokkuuden parantamisen ja ympäristövaikutusten pienentämisen tärkeyttä ilmaston, luonnonvarojen ja laskentatehon tarpeen osalta. Näiden lisäksi tutustutaan konseptiin vihreästä laskennasta. Luvussa 3 käsitellään muutaman tapausesimerkin avulla, miten energiatehokkuus on kehittynyt komponenttien, ja erityisesti prosessorien osalta. Luku 4 keskittyy erilaisiin tekniikoihin, joiden avulla energiatehokkuutta voidaan parantaa ja ympäristövaikutuksia pienentää. Luvussa esitellään keskitettyjä ratkaisuja, joita voitaisiin hyödyntää etenkin tiloissa ja toimistoissa, joissa on paljon yksittäisiä keskusyksiköitä. Lopuksi pohditaan, voisiko näiden tekniikoiden ja uusien energiatehokkaiden komponenttien avulla jopa luopua perinteisistä keskusyksiköistä niitä paljon hyödyntävissä tiloissa ja toimistoissa.

2. Keskusyksiköiden energiatehokkuuden tärkeys

Energiatehokkuus eli hyvä hyötysuhde takaa, että keskusyksikkö kuluttaa mahdollisimman vähän energiaa suorittaessaan jonkin sille annetun tehtävän mahdollisimman tehokkaasti ja pienimmällä mahdollisella hukkaenergian määrällä. Hyvä energiatehokkuus on peruslähtökohta nykyaikaisen laskentayksikön suunnittelussa. [Castro *et al.* 2012]

Kolmen vuoden ikäinen tavallinen keskusyksikkö voi kuluttaa *tyhjäkäynnillä* (idle desktop), eli kun se on käynnissä ja käyttämättömänä, jopa 60 % sen huipputehosta [Bila *et al.* 2012]. Tämä lukema on varsin suuri, ja sen aiheuttamista kuluista ja vaikutuksista kärsii moni taho. Jo pelkästään rahassa mitattuna useamman tyhjäkäynnillä olevan keskusyksikön sisältämä tietokonesali voi kustantaa ylläpitäjälle vuodessa merkittäviä summia. Ilman energiatehokkaita ja oikein mitoitettuja laskentayksiköitä kulutamme turhaan maapallon rajallisia resursseja.

Wang ja Khan [2011] määrittelevät ympäristövaikutukset-termin tarkoittamaan datakeskusten yhteydessä erilaisia vihreyteen ja ympäristöystävällisyyteen viittaavia parametreja, kuten energiankulutusta ja hiilidioksidipäästöjä.

Heidän määritelmäänsä voidaan kuitenkin soveltaa myös keskusyksiköihin. Kun pystytään tuottamaan ja käyttämään kokonaisuudessaan hyvän energiatehokkuuden omaavia keskusyksiköitä, ympäristövaikutukset laskevat samalla. Tarkastelun ulkopuolelle jätetään seikat, jotka ovat käyttäjän itsensä päätettävissä, kuten miten tuotettua sähköä hän hankkii.

Seuraavissa kohdissa tarkastellaan energiatehokkuutta ja ympäristövaikutuksia erilaisista näkökulmista, esitellään joitakin keinoja niiden parantamiseksi sekä tutustutaan tutkielman myöhemmissäkin luvuissa käytettyyn pieneen aineistoon prosessorien kehitykseen liittyen.

2.1. Laskentatehon tarjonta ja tarve

Berl ja Meer [2010] tutkivat tavallisissa toimistoissa käytettyjä tietokoneita; heidän mukaansa ne ovat vähintäänkin alikäytettyjä. Käytännössä tämä tarkoittaa, että näiden tietokoneiden koko laskentatehoa ei hyödynnetä, ja kuten aiemmin todettiin, vuoden 2012 keskusyksikkö voi kuluttaa jopa 60 % sen huipputehosta, vaikka sillä ei tehtäisikään mitään.

Tietokoneiden prosessorien laskentateho on kasvanut kovaa vauhtia koko 2000-luvun. Keräsin taulukkoon 1 kolmen suurin piirtein toisiaan vastaavan eri ikäluokan prosessorin tiedot, joilla voidaan arvioida laskentatehon määrää ja sen tarvetta. Prosessorit vastaavat tyypillisiä tavallisissa toimistoissa käytettyjä keskusyksiköiden prosessoreja.

MALLI	JULKAISUVUOSI	JULKAISUHINTA $	LASKENTATEHO
CORE2 DUO E4400	2007	110,00	1142
CORE I3-2100	2011	120,00	3632
CORE I3-6100	2015	117,00	5522

Taulukko 1. Prosessorien laskentatehon kehitys. Laskentatehon yksikkö on Passmark-testin antamat pisteet [Passmark]. Prosessorit [Intel].

Taulukosta 1 nähdään, että laskentateho on kasvanut voimakkaasti ja suhteellisen lineaarisesti näiden mallien välillä. Tavallisten toimistoissa käytettyjen ohjelmistojen vaatima suorituskyky ei ole kuitenkaan kasvanut samassa suhteessa. Varsinkin vuosien 2011 ja 2015 välillä kasvu on ollut hyvin marginaalista, ja oman kokemukseni mukaan vuoden 2011 CORE I3 -prosessori on tarpeeksi tehokas näiden ohjelmistojen suorittamiseen jopa niin, että suurin osa prosessorin laskentatehosta on silti käyttämättömänä.

Näissä prosessoreissa ei ole kuitenkaan kehittynyt vain laskentateho, vaan niistä on samalla tullut huomattavasti energiatehokkaampia, mistä lisää luvussa 3. Lisäksi prosessoreihin *integroidut näytönohjaimet* (integrated GPU) ovat jo niin tehokkaita, ettei tavalliseen toimistokeskusyksikköön tarvitse lisätä erillistä

näytönohjainkomponenttia [Scogland *et al.* 2010]. Nämä integroidut näytönohjaimet ovat kuitenkin vielä liian tehottomia, jos keskusyksiköllä suoritetaan näytönohjainta paljon kuormittavia työtehtäviä.

2.1.1. Vihreä laskenta

Vuonna 1992 Yhdysvaltojen ympäristösuojeluvirasto aloitti Energy Star -ohjelman. Siitä kehittyi kansainvälinen energiatehokkaiden ja pääasiassa tietoteknisten laitteiden standardi. Tämän standardin pohjalta on syntynyt myös *vihreän laskennan* (Green computing) malli. Se pyrkii ottamaan huomioon paineen ympäristönsuojelun ja energiansäästön osalta. Se kannustaa vähentämään jatkuvaa komponenttien uusimista ja hyödyntämään jo olemassa olevien komponenttien suorituskykyä, jotta niiden energiatehokkuus pysyisi hyvällä tasolla ja niistä saataisiin tarjolla oleva suorituskyky irti. [Hu *et al.* 2011]

2.1.2. Vihreä tietotekniikka

Vihreän laskentamallin tavoitteen voi sisällyttää *vihreän tietotekniikan* (Green IT) osatavoitteeksi. Vihreällä tietotekniikalla tarkoitetaan kaikkea tietotekniikkaa, joka pyrkii säästämään energiaa ja vähentämään kasvihuonekaasupäästöjä. Sen periaatteisiin kuuluu, ettei tietotekniikkaa myöskään uusita pelkästään uusimisen vuoksi. Näin ollen vihreää tietotekniikkaa hyödynnetään jouduttamaan kehitystä kohti kestävämpiä ja ympäristöystävällisempiä yhteiskuntia. [Trimi and Park 2012] Trimin ja Parkin mukaan vihreällä tietotekniikalla on keskeinen rooli myös materiaalien kierrätyksen ja kiertotalouden kehityksessä, joka osaltaan helpottaa ilmastoomme kohdistuvaa kuormitusta.

2.2. Energiatehokkuus ilmaston kannalta

Ilmasto kuormittuu aina käytettäessä ja valmistettaessa keskusyksiköitä. Valmistusvaiheessa erilaiset valmistusmenetelmät ja valinnat energianlähteiden välillä vaikuttavat kuormituksen määrään, jolla taas on suuri merkitys tuotannon ympäristöystävällisyyteen. Energiatehokkuuden kannalta tuotannon ympäristöystävällisyydellä ei ole niin suurta merkitystä, mutta kokonaisuutena tuotantokin on osaltaan vastuussa kuluttajien ohjaamisesta energiatehokkaampien tuotteiden pariin kestävän kehityksen mukaisesti.

WCED (World Commission of Environment and Development) muotoilee kestävän kehityksen suurin piirtein näin: tarvittava kehitys nykyiselle yhteiskunnalle ilman kompromisseja tulevien sukupolvien kustannuksella [Trimi and Park 2012]. Ei siis ole järkevää uusia vanhoja hyvin tehtävänsä suorittavia keskusyksiköitä vain uusimisen takia, koska se tuottaa turhaan päästöjä ja kuormittaa ympäristöä.

Energiatehokkuus on kuitenkin erittäin tärkeää siltä kannalta, etteivät käytössä olevat keskusyksiköt kuluttaisi liikaa energiaa turhaan vain ollakseen päälle kytkettyinä ja tuottaen samalla välillisesti päästöjä energiantuotannon kautta. Vihreän laskentamallin mukaan keskusyksiköiden tulisi olla käytössä mahdollisimman tehokkaasti.

Vihreää laskentamallia tukee eteläamerikkalainen UnaGrid-niminen virtuaalinen infrastruktuuriverkko. Sen toiminta perustuu paikoitellen lyhytaikaiseenkin tietokonesalien käyttämättömän laskentatehon hyödyntämiseen [Castro *et al.* 2012, 547]. Tämä verkko on erityisen hyvä ilmaston ja ympäristöystävällisyyden puolesta. Kohteissa, joissa verkko on käytössä, keskusyksiköitä on todella paljon. Kun nämä keskusyksiköt ja niiden laskentateho yhdistetään UnaGridillä hyötykäyttöä ja palvelinkeskustasoista laskentaa varten, säästetään helposti resursseja sekä energiaa.

Kokonaisuutena keskusyksiköiden energiatehokkuus ja niiden tuottamien ympäristövaikutusten minimointi on tärkeää ilmastonmuutoksen kehityksen katkaisemiseksi ja kestävän kehityksen turvaamiseksi myös tuleville sukupolville.

2.3. Luonnonvarojen yhteys energiatehokkuuteen

Vielä tällä hetkellä jokaisen tietoteknisen tuotteen valmistukseen kuluu huomattava määrä uusiutumattomia luonnonvaroja, mikä ei ole kestävän kehityksen mukaista [Trimi and Park 2012]. Tästäkin huolimatta, keskusyksiköt ja niistä sovelletut tietotekniset ratkaisut ovat edelleen laskentateholtaan ja hintatasoltaan järkevin ratkaisu yksittäiselle kuluttajalle.

Vaikka Suomessa tietotekniikan kierrätys on jo hyvällä tasolla, kuluu jokaisen keskusyksikön valmistamiseen ja ylläpitoon ympäri maailmaa uusiutumattomia ja kierrätyksen kautta hyödyntämättömiä luonnonvaroja. Viimeisen neljän vuoden aikana, varsinkin laskentatehon kehityksen ja useimpien toimisto-ohjelmien suorituskykyvaatimusten kasvaessa eri tahtia, ovat komponenttivalmistajat keskittyneet suoremmin keskusyksiköiden komponenttien energiatehokkuuden parantamiseen [Trimi and Park 2012]. Tämä on vähentänyt tavallisten kuluttajien tarvetta investoida uusiin komponentteihin ja keskusyksiköihin [McMillan 2015].

Energiatehokkaat ja hyvän vähitellen kasvavan laskentatehon omaavat komponentit mahdollistavat vihreän tietotekniikan määrän ja hyödyntämisen kasvun. Nämä komponentit tukevat osaltaan kestävän kehityksen periaatetta ja mahdollistavat samalla niitä valmistaville yrityksille profiloitumisen vihreänä ja ympäristöystävällisenä valmistajana [Trimi and Park 2012]. Profiloituminen edellä kuvatulla tavalla auttaa valmistajia paikkaamaan komponenttien myyn-

nissä näkyvää laskua ihmisten investoidessa ympäristöystävällisempiin vaihtoehtoihin [Clark 2015].

Vihreäksi kasvuksi (Green growth) kutsutaan taloudellisen kasvun mallia, jossa kasvu perustuu vihreiden tuotteiden ja palveluiden tuottamiin uusiin työpaikkoihin [Zysman and Huberty 2012]. Zysmanin ja Hubertyn mukaan vielä kolme vuotta sitten, vuonna 2012, vihreää kasvua ja investointeja siihen pidettiin yleisesti kestämättömänä ja vain väliaikaisena vaiheena. He pyrkivät kuitenkin vakuuttamaan, että vihreä kasvu luo uusia rakenteita nykyiseen yhteiskuntaamme ja sitä kautta synnyttää vähitellen uudenlaisia taloudellisia hyötyjä. Tästä ensimmäiset merkit ovat jo nähtävissä kuluttajien hankkiessa uusimpia keskusyksiköiden komponentteja, jotka ovat ennen kaikkea energiatehokkaampia kuin vanhat edelleenkin riittävää laskentatehoa tarjoavat (vertaa taulukko 1). Tämä komponenttivalmistajien tuotekehityksen suunta tukee osaltaan kestävää kehitystä ja muun muassa ainakin luonnonvarojen välillistä säästöä, sillä niitä ei kulu niin paljoa näiden vihreiden keskusyksiköiden ylläpitoon.

3. Energiatehokkuuden kehitys komponenttien osalta

Tutkielman toisessa luvussa käsiteltiin sitä, miksi energiatehokkuuden huomioiminen on tärkeää ja miten se on kehittynyt lähivuosina. Energiatehokkuuden parantaminen vaikuttaa välillisesti myös keskusyksiköiden aiheuttamiin ympäristövaikutuksiin. Valmistajat ovat huomanneet energiatehokkaiden ja ympäristöystävällisten komponenttien potentiaalin [Trimi and Park 2012].

Komponenttivalmistajien huomattua potentiaalin, jota vihreämpi tietotekniikka tuo, ovat ne alkaneet kehittää ja tuottaa uusia tekniikoita, joiden avulla näiden komponenttien energiatehokkuutta on voitu kehittää entistä paremmaksi [Zysman and Huberty 2012]. Komponenttien energiatehokkuus onkin kehittynyt merkittävästi suhteessa suorituskykyyn viimeisen kahdeksan vuoden aikana.

Tässä luvussa käsitellään ensin komponenttien energiatehokkuuden kehitystä yleisellä tasolla ja esitellään joitakin komponenttivalmistajien merkittäviä tekniikoita, joilla tätä on parannettu. Tämän jälkeen käsitellään, miten energiatehokkuus on kehittynyt suhteessa suorituskykyyn, ja lopuksi paneudutaan vielä lyhyesti komponenttien hintatason kehitykseen suhteessa energiatehokkuuteen.

3.1. Energiatehokkuuteen pyrkivät tekniikat komponenteissa

Komponenttivalmistajien painopiste on siirtynyt yhä energiatehokkaampien ratkaisujen muodostamiseen. Ne ovat kehittäneet paljon erilaisia tekniikoita, joilla pyritään säästämään energiaa varsinkin sellaisina hetkinä, kun komponentit ovat pienellä kuormituksella tai lepotilassa. Näiden tekniikoiden oheis-

sa komponenttivalmistajat ovat siirtyneet myös yhä energiatehokkaampiin suoritinarkkitehtuureihin.

Osa komponenttivalmistajien kehittämistä ratkaisuista hukkaenergian minimoimiseksi ja energiatehokkuuden parantamiseksi ei ole kuitenkaan onnistunut. Esimerkkeinä näistä epäonnistuneista tekniikoista voidaan pitää virransäästötiloja ja Wake-on-LAN:ia [Bila *et al.* 2015]. Erilaiset virransäästötilat eivät ole olleet onnistuneita, sillä käyttäjät ovat useimmiten laittaneet ne pois käytöstä paremman vasteen saamiseksi tietokoneiltansa. Wake-on-LAN ei ole ollut onnistunut, sillä se ei oikeastaan toteuta sille suunniteltua tehtävää. [Bila *et al.* 2015]

Wake-on-LAN:in tarkoitus on mahdollistaa tietokoneen herättäminen lepotai horrostilasta sekä kokonaan sammuksista. Se epäonnistuu kuitenkin, sillä menetelmä on suhteellisen energiatehoton. Jotta tietokone voisi herätä näiden verkkokutsujen avulla, täytyisi sen suorittimen vastaanottaa näitä kutsuja, jolloin todellisuudessa tietokone on itseasiassa päällä koko ajan hukaten energiaa. [Bila *et al.* 2015]

Vaikka käyttäjällä onkin suuri vaikutus energiatehokkaiden tekniikoiden tarkoituksen mukaiseen toimintaan, ovat komponenttivalmistajat kuitenkin tuoneet markkinoille myös hyvin toimivia ja keskusyksiköiden energiatehokkuutta merkittävästi parantavia tekniikoita. Nämä tekniikat eivät ole niinkään riippuvaisia käyttäjän tekemistä valinnoista, vaan toimivat automaattisesti ja dynaamisesti tietokoneessa komponenttitasolla.

3.1.1. Dynaaminen taajuuden skaalaus

Dynaaminen taajuuden skaalaus (Dynamic frequency scaling, DFS) on sisällytetty käytännössä kaikkiin moderneihin suorittimiin. Sen avulla suorittimet voivat nostaa ja laskea kellotaajuuttaan dynaamisesti niiltä vaaditun työmäärän mukaan annettujen maksimiarvojen puitteissa. [Castro *et al.* 2012] Tämä on yksittäisenä tekniikkana yksi merkittävimmistä prosessorien ja näytönohjainten energiatehokkuutta nostavista tekniikoista.

3.1.2. Dynaaminen virranhallinta

Dynaaminen virranhallinta (Dynamic power management DPM) pyrkii minimoimaan suorittimelle syötetyn sähkön määrän juuri sopivaksi. Tekniikan tarkoitus on vähentää kuluvan hukkaenergian määrää ja samalla laskea suorittimen lämmöntuottoa. [Castro *et. al.* 2012]

Käytännössä tekniikka on välttämätön, jotta dynaamista taajuuden skaalausta voitaisiin hyödyntää sen koko potentiaalin osalta. Dynaaminen virranhallinta antaa aina oikean määrän sähköä, jonka suoritin tarvitsee suorittaakseen jonkin tehtävän.

3.1.3. AMD ZeroCore Power

Komponenttivalmistaja *AMD* (Advanced Micro Devices) on kehittänyt ZeroCore Power -tekniikan. Sen avulla erityisesti tietokoneissa, joissa on prosessoriin integroitu näytönohjain sekä erillinen näytönohjain, voidaan sulkea tämä toinen erillinen näytönohjain lähes täysin. Tämä mahdollistaa näytönohjaimen sähkönkulutuksen pudottamisen tyhjäkäynnillä jopa 20 %:iin aiemmasta tyhjäkäynnin sähkönkulutuksesta. [Smith 2011]

Tekniikka säästää merkittävästi sähköä ja lisää varsinkin tehotyöasemien, joissa useimmiten on erillinen näytönohjain, energiatehokkuutta ja ympäristöystävällisyyttä huomattavasti. ZeroCore Power -tekniikkaa sovelletaan myös integroidun näytönohjaimen sisältämiin AMD:n prosessorilla varustettuihin tietokoneisiin. Näissä integroitu näytönohjain asetetaan vastaavaan tilaan tietokoneen siirtyessä lepotilaan. Tekniikkaa käytetään myös työasemissa, joissa on esimerkiksi yli kaksi näytönohjainta, jolloin vain tarvittava määrä näytönohjainten laskentatehosta pidetään dynaamisesti saatavilla ja muihin sovelletaan ZeroCore Power -tekniikkaa.

3.2. Energiatehokkuuden kehitys suhteessa suorituskykyyn

Keskusyksiköiden laskentatehoa mitataan useimmiten jonkin jotain tiettyä *työtehtävää mallintavan testin* (benchmark) avulla. Näiden laskentatehoon keskittyvien testien rinnalle on kuitenkin otettu myös laskentatehon ja kulutetun sähkön suhdetta kuvaava luku, joka varsinkin suurissa tietokonesaleissa ja palvelinkeskuksissa on yksi merkittävimmistä suunnittelulähtökohdista. [Huppler 2012, 61] Selvyyden ja vertailuvuuden vuoksi käsitellään energiatehokkuuden kehittymistä suhteessa suorituskykyyn käyttämällä jo luvussa 2 esiteltyjä kolmea prosessoria esimerkkeinä taulukossa 2.

MALLI	JULKAISUVUOSI	LASKENTATEHO	TDP	LASKENTATEHO/TDP
CORE2 DUO E4400	2007	1142	65	17,6
CORE I3-2100	2011	3632	65	55,9
CORE I3-6100	2015	5522	51	108,3

Taulukko 2. Prosessorien laskentatehon ja sähkönkulutuksen suhteen kehitys. Laskentatehon yksikkö on Passmark-testin antamat pisteet [Passmark]. Prosessorit [Intel].

Taulukossa 2 kuvataan sähkönkulutusta yleisesti käytössä olevalla *TDP-arvolla* (Thermal design power). Tämä tarkoittaa suurinta sähkönkulutuksen määrää, jonka prosessorin jäähdytyksen täytyy pystyä käsittelemään normaalioloissa. Komponenttivalmistajat antavat nämä arvot tuotteillensa [Intel].

Taulukon 2 perusteella nähdään, että vuosien 2007 ja 2015 välillä prosessorien energiatehokkuus on kasvanut valtavaa vauhtia. Kehitys on ollut hyvin merkittävää, vaikkakin taulukon 2 perusteella nähdään myös, että laskentatehon kehityksen hidastuessa vuodesta 2011 vuoteen 2015 on myös energiatehokkuuden kehitys hidastunut. Taulukon 2 mukainen kehitys on pystytty saavuttamaan komponenttivalmistajien kehittäessä käyttäjän normaalista toiminnasta riippumattomia tekniikoita ja tehokkaampia suoritinarkkitehtuureja. Kuten Bila ja muut [2015] esittivät, ei valintaa energiatehokkaan ja energiaa hukkaavan tietotekniikan välillä kannata jättää käyttäjän vastuulle.

Valmistajien kehitettyä paljon erilaisia uusia tekniikoita energiatehokkuuden parantamiseksi ovat komponenttivalmistajat pyrkineet profiloitumaan voimakkaastikin energiatehokkaiksi ja esimerkiksi pitkiä työskentelyaikoja akkuvirralla tarjoavien ratkaisujen rakentajiksi. Esimerkiksi Intel on markkinoinut kaikista energiatehokkaimpia mobiiliprosessoreitansa Ultrabook^{tm}-tietokoneina [Intel]. Näissä malleissa yhdistyy erinomainen energiatehokkuus ja tavalliseen toimistotyöhön täysin riittävä suorituskyky.

3.3. Energiatehokkaan laskentatehon hintakehitys

Vaikka valmistajat ovatkin pyrkineet profiloitumaan ja kohottamaan imagoaan energiatehokkaiden komponenttien valmistajina, ei tämä ole kuitenkaan nostanut komponenttien hintatasoa yleistä hintatason nousua nopeammin. Energiatehokkaan perusratkaisun voi hankkia samalla hinnalla kuin vähemmänkin energiatehokkaan, samantasoista suorituskykyä tarjoavan.

MALLI	JULKAISUHINTA $	LASKENTATEHO	LASKENTATEHO/ JULKAISUHINTA $
CORE2 DUO E4400	110,00	1142	10,4
CORE I3-2100	120,00	3632	30,3
CORE I3-6100	117,00	5522	47,2

Taulukko 3. Prosessorien laskentatehon hintakehitys. Laskentatehon yksikkö on Passmark-testin antamat pisteet [Passmark]. Prosessorit [Intel].

Taulukosta 3 nähdään, ettei energiatehokkuuden kasvulla ja prosessorien hintatason kehityksellä ole oikeastaan riippuvuutta.

Suurin osa keskusyksiköistä on kuitenkin edelleen alikäytettyjä [Berl and Meer 2010]. Näin ollen ne kuluttavat turhaan sähköä, komponenttivalmistajien kehittämistä tekniikoista huolimatta. Tämä nostaa luonnonvarojen kulutusta ja keskusyksiköiden käytön ympäristövaikutuksia. Olisikin perusteltua, että näiden alikäytettyjen resurssien sijaan olisi tarjolla tarpeeseen mukautuvaa energiatehokasta laskentatehoa, jota voitaisiin hyödyntää muussa käytössä, kun yksittäinen käyttäjä ei sitä tarvitse [Berl and Meer 2010].

4. Energiatehokkuutta parantavat ulkopuoliset tekniikat

Jotta energiatehokkuutta voitaisiin entisestään parantaa ja ympäristövaikutuksia pienentää, tulisi laskentayksiköiden suunnittelun perustana olla vihreän laskentamallin hyödyntäminen ja soveltaminen. Vihreän laskennan periaatteiden mukaan laskentayksiköiden, palvelimien ja tässä tapauksessa erityisesti keskusyksiköiden tulisi olla hyötysuhteeltaan hyviä, ja niiden ei tulisi haaskata resursseja [Hu *et al.* 2011]. Trimi ja Park [2012] linjaavat, että ollakseen vihreää tietotekniikan tulisi tuottaa mahdollisimman vähän kasvihuonekaasuja sekä säästää energiaa, ja sen valmistuksen tulisi olla mahdollisimman ympäristöystävällistä.

Miller ja Pegah [2007] esittävät, että jakamalla resursseja useamman käyttäjän kesken voidaan pidentää ikääntyvien komponenttien elinkaarta huomattavasti. Tätä he vertaavat tilanteeseen, jossa laskentayksikkö olisi erillinen kokonaisuutensa ja omaisi vain sisältämänsä laskentatehon. Resurssien jakamisella tarkoitetaan karkeasti mallia, jossa yksittäinen käyttäjä voi tarpeen tullen hyödyntää muille käyttäjille varattua laskentatehoa lähiverkon yli [Berl and Meer 2010].

Jaettujen resurssien avulla pystytään pienentämään myös ympäristövaikutuksia, sillä keskusyksiköiden komponenttien tuottamiseen ei kulu niin paljon resursseja pidentyneen elinkaaren vuoksi [Miller and Pegah 2007]. Tämä antaa myös komponenttivalmistajille aikaa kehittää ympäristöystävällisempiä valmistusprosesseja. Keventyneiden ympäristövaikutusten lisäksi jaetuilla resursseilla ja järkevällä virranhallinnalla voidaan säästää huomattavia määriä energiaa [Berl and Meer 2010]. Jaettujen resurssien hyödyntäminen mahdollisuuksien mukaan palveleekin juuri vihreän tietotekniikan ja vihreän laskentamallin periaatteita.

Tässä luvussa käsitellään keinoja ja tekniikoita, joilla varsinkin useita keskusyksiköitä sisältävien tietokonesalien tai tilojen energiatehokkuutta voidaan parantaa entisestään. Luvussa esitettävät keinot ja tekniikat pohjautuvat resurssien jakamisesta saatuihin hyötyihin ja etuihin.

4.1. Virtualisointi

Virtualisoinnilla (Virtualization) tarkoitetaan jonkin palvelun tai käyttäjän ympäristön erotusta fyysisistä laskentaresursseista, jotka itse asiassa toteuttavat ja tarjoavat tämän kyseisen palvelun [Yang *et al.* 2011, 282-283]. Virtualisoinnin avulla tuotetussa ympäristössä eri käyttäjät eivät myöskään pääse käsiksi toistensa virtuaalisiin lohkoihin fyysisestä resurssista. Virtualisoitu ympäristö ei ole myöskään sidottu johonkin tiettyyn fyysiseen resurssiin, vaan se voi käyttää suhteellisen dynaamisesti eri resursseja, joita muutkin käyttäjät käyttävät, esimerkiksi levytilaa ja suoritinta. [Yang *et al.* 2011].

Virtuaalikoneet (Virtual machine) ovat virtualisointia hyödyntäviä virtuaalisia ympäristöjä, joissa voi suorittaa ohjelmia ja tehtäviä kuten tavallisessakin yksittäisessä tietokoneessa [Yang et al. 2011]. Virtuaalikoneiksi kutsutaan virtualisoinnin avulla tuotettuja ympäristöjä, joissa varsinkin palvelimien puolella useat kymmenetkin käyttäjät voivat käyttää samanaikaisesti samalta fyysiseltä resurssilta toimivaa yksittäisten virtuaalikoneiden joukkoa. Nämä virtuaalikoneet näyttäytyvät näissä tilanteissa yksittäisinä tietokoneina käyttäjille. Virtuaalikonetta ja nimenomaan *virtuaalityöpöytää* (Virtual desktop) voidaan käyttää samaan tapaan kuin normaalia tietokonetta. Käyttäjän oma tietokone toimii näyttönä ja ohjaimena, johon virtuaalikone lähettää ja laskee sisällön ja jolla kontrolloidaan sisältöä normaaliin tapaan [Berl and Meer 2010].

Virtuaalikoneita pidetään luonnollisena tapana käyttäjän ja laskentaympäristön erottamisessa toisistaan [Castro *et al.* 2012]. Virtuaalikoneiden avulla tavallisilta tietokoneilta ei ole välttämätöntä vaatia niin suuria ponnistuksia energiatehokkuuden ja virransäästön osalta. Ne eivät nimittäin kuormitu esittäessään käytännössä vain suoratoistovideota virtuaalikoneella tapahtuvista asioista, jolloin ne voivat olla varsinkin uudempien prosessorien tapauksessa käytännössä lepotilassa laskennan hoituessa virtuaalikoneella. Laskentateholtaan heikko useiden vuosien ikäinen keskusyksikkö varustettuna esimerkiksi taulukon 1 CORE2 DUO E4400 -prosessorilla on kyllin tehokas toimimaan virtuaalityöpöydän vastaanottavana ja käyttäjälle näyttävänä tietokoneena.

Millerin ja Pegahin [2007] mukaan virtualisoinnin avulla voidaan pidentää vanhemman tai heikkotehoisen tietokoneen elinkaarta huomattavasti. Heidän mukaansa tietokone voisi toimia vain näyttönä ja ohjaimena, jolloin varsinainen laskenta ja ohjelmien suoritus tapahtuisi itse asiassa jollakin virtuaalikoneella yhdessä keskitetyssä laskentayksikössä. Tämä malli säästäisi huomattavasti luonnonvaroja, sillä jopa pieni yksityinen palvelin pystyy tarjoamaan suurelle joukolle keskusyksiköitä virtuaalisen työskentely-ympäristön [Bila *et al.* 2012]. Esimerkiksi vuodelta 2009 oleva Intel CORE i7-960 -prosessori [Intel] oli suu-

rimman osan ajasta alle 40 % kuormituksella, kun käytimme Projektityökurssilla virtuaalityöpöytiä ohjelmistotuotantoon viiden opiskelijan voimin.

Virtualisoinnin tuomat hyödyt eivät kuitenkaan rajoitu pelkästään energiatehokkuuden paranemiseen, ympäristövaikutusten lievenemiseen ja taloudellisiin hyötyihin, jotka syntyvät, kun tarve keskusyksiköiden komponenttien jatkuvalle uusimiselle vähentyy. Binderin ja Surin [2009] mukaan virtualisoinnin avulla pystytään myös tarjoamaan käyttäjälle juuri oikea määrä laskentatehoa käyttäjän sitä tarvitessa. Tarvittaessa ajankohtana, jolloin tämän keskitetyn laskentayksikön, eli useiden virtuaalikoneiden, fyysiset resurssit olisivat kevyemmällä kuormituksella, voitaisiin sitä tarvitseville käyttäjille tarjota erittäin voimakasta laskentatehoa.

Virtualisoinnin, jaettujen resurssien ja järkevän virranhallinnan avulla voidaan säästää huomattavia määriä energiaa. Berlin ja Meerin [2010] mukaan tämä ratkaisu voisi tuoda parhaimmillaan jopa 75 % energiasäästöt verrattuna yksittäisiin keskusyksiköihin, joissa tapahtuisi kaikki laskenta. Virtualisointi tuo myös mahdollisuuden käyttämättömien fyysisten resurssien sammuttamiseen, jos ne ovat turhaan päällä. Binderin ja Surin [2009] mukaan tämä ei vaikuta käyttäjälle tuotettujen palveluiden ja työskentely-ympäristön käyttöön tai käytännön suorituskykyyn juuri lainkaan. Heidän mukaansa nämä resurssit voidaan taas ottaa käyttöön dynaamisesti, kun tarve niille kasvaa.

Seuraavassa kohdassa tarkastellaan vielä hieman tarkemmin jo edellä mainittuja virtuaalityöpöytiä sekä Thin-client-mallia. Virtuaalityöpöytiä käsitellään energiatehokkuuden ja niiden tuomien mahdollisuuksien osalta, Thin-client-mallia taas tapana toteuttaa tämä virtualisoitu työskentely-ympäristö.

4.2. Etätyöpöydät

Virtuaalityöpöytiä voidaan käyttää nostamaan energiatehokkuutta yhdessä virtuaalikoneiksi jaetun palvelimen, keskitetyn laskentayksikön, kanssa. Virtuaalityöpöytäratkaisut ovat myös erittäin kustannustehokkaita investointisummien jäädessä suhteellisen mataliksi infrastruktuurin rakentamisen jälkeen, ja niiden mahdollistaessa myös hyvän suorituskyvyn niin grafiikan kuin hyötykäytön osalta [Miller and Pegah 2007].

Virtuaalityöpöydät mahdollistavat myös grafiikan tuottamisen etänä, josta tarkemmin kohdassa 4.3, jolloin virtuaalikoneiden avulla pystytään tarjoamaan myös graafisesti vaativaa materiaalia käyttäjille [Miller and Pegah 2007]. Virtuaalityöpöytien vasteajatkaan eivät modernien verkkotekniikoiden takia ole juurikaan korkeammat verrattuna tavallisiin keskusyksiköiden työpöytiin. Näin ollen eroa virtuaalityöpöydän ja tavallisen omalla tietokoneella suoritetun ympäristön välillä on jopa hankala huomata.

Virtuaalityöpöydät mahdollistavat energiatehokkaan järjestelmän myös sen ollessa kevyellä kuormalla tai jopa tyhjäkäynnillä, sillä virtuaalityöpöydän käyttäjän puolen täytyy suoriutua vain todella kevyestä laskennasta. Sen ei tarvitse tehdä muuta kuin tulkita käyttäjän liikkeitä ja renderöidä näytölle virtuaalityöpöydällä luotu grafiikka. [Bila *et al.* 2015]

Virtuaalityöpöytiin perustuvat ratkaisut eivät ole olleet kuitenkaan kovinkaan suosittuja, ja useimmissa tapauksissa onkin valittu ennemmin normaali keskusyksikkö tällaisen Thin-client-keskusyksiköistä ja tehokkaasta keskitetystä laskentayksiköstä koostuvan järjestelmän sijaan [Bila *et al.* 2015]. Yksittäisistä keskusyksiköistä koostuvat ryppäät eivät ole kuitenkaan yhtä energiatehokkaita, sillä keskusyksiköt eivät säästä energiaa yhtä tehokkaasti verraten virtuaalityöpöytien ja Thin-client-keskusyksiköiden avulla toimiviin järjestelmiin.

Virtuaalityöpöydät mahdollistavat myös keskitetyn laskentayksikön tuottaman hukkalämmön hyödyntämisen muissa yhteyksissä, esimerkiksi kohdennetusti kaukolämmön tuoman veden lämmittämisessä entisestään. Virtuaalityöpöydät tukevat omalta osaltaan vihreän laskentamallin ja vihreän tietotekniikan periaatteita monin tavoin. Ne tarjoavat ratkaisuja juuri näiden esittämiin haasteisiin [Trimi and Park 2012].

Virtuaalityöpöydät ovat ratkaisuina hyviä myös ympäristövaikutusten osalta, sillä nämä hyvinkin pienet ja luonnonvaroja säästävät Thin-client-tietokoneet voidaan koostaa vanhoiksi käyneistä yksittäisistä keskusyksiköistä. Tällöin saadaan hyödynnettyä näitä jo kerran jalostettuja luonnonvaroja entistä tehokkaammin, jolloin aiemmille investoinneillekin saadaan enemmän katetta [Zysman and Huberty 2012].

Thin-client-tietokoneet, eli kevyet tai pienet tietokoneet, ovat tietokoneita, joiden tehtävänä on usein toimia virtuaalityöpöytäyhteyden käyttäjän puoleisena laitteena. Nämä tietokoneet ovat useimmiten fyysiseltä kooltaan murto-osan normaalista keskusyksiköstä. Niiden suorituskyky on yleisesti hyvin vaatimatonta. [Berl and Meer 2010]

Kuten jo aiemmin todettiinkin, Thin-client-tietokoneiden tehtävänä on vain ottaa käyttäjän komennot vastaan ja lähettää ne palvelimelle, keskitetylle laskentayksikölle, jolla virtuaalityöpöytäkin on. Tämä palvelin sitten tuottaa grafiikan, laskee ja suorittaa ympäristössä tapahtuvat asiat ja lähettää ne takaisin tälle Thin-client-tietokoneelle, joka renderöi ne käyttäjän nähtäville.

Thin-client-tietokoneet perustuvat siis vihreän tietotekniikan periaatteita noudattavaan konseptiin [Trimi and Park 2012]. Pienet, energiatehokkaat ja vähän luonnonvaroja kuluttavat tietokoneet ovat erinomainen vaihtoehto energiatehokkuuden parantamiseen ja ympäristövaikutusten hillintään. Näitä koneita ei tarvitse uusia kovinkaan usein, sillä virtuaalityöpöytien suorittaminen on suhteellisen kevyttä.

Kuitenkin tarvittaessa voimakkaampaa graafista suorituskykyä voidaan turvautua grafiikan etärenderöintiin. Tätä voidaan hyödyntää esimerkiksi tilanteessa, jossa tällä Thin-client-tietokoneella suoritettaisiin muita, virtuaalityöpöytäyhteyden ulkopuolisia tehtäviä.

4.3. Etärenderöinti

Etärenderöinti (Remote rendering) tarkoittaa 3D-grafiikan renderöintiä jossakin erillisessä laskentayksikössä ja sen tuottaman kuvan näyttämistä toisessa tietokoneessa [Shi and Hsu 2015]. Thin-client-käyttöön verraten tämä 3D-grafiikan renderöinti voitaisiin suorittaa esimerkiksi samassa keskitetyssä laskentayksikössä kuin itse virtuaalityöpöytäkin. Tämän kaltainen lähestymistapa tuo useita etuja verrattuna tilanteeseen, jossa hyvää graafista suorituskykyä tarvitsevat sovellukset pitäisi suorittaa omilla erillisillä keskusyksiköillänsä.

Erillisin keskusyksiköin toteutetussa järjestelmässä nämä keskusyksiköt tarvitsisivat voimakkaita erillisiä näytönohjaimia suoriutuakseen tehtävistään. Jos nämä tehtävät olisivat hyvin raskaita, kuten esimerkiksi sisällöntuotantoa johonkin suoratoistopalveluun, kuluttaisivat nämä yksittäiset keskusyksiköt kohtuuttomia määriä energiaa, ja samalla niiden lepotilan ja tyhjäkäynnin energiatehokkuus laskisi [Scogland *et al.* 2010].

Etärenderöinnin suorittava laskentayksikkö (Rendering server) ottaa myös virtuaalityöpöytiä hallitsevien laskentayksiköiden tapaan käyttäjän hallintalaitteiden komentoja vastaan. Etärenderöinti tarjoaa joustavaa ja erittäin hyvällä hyötysuhteella varustettua laskentatehoa käyttäjillensä. [Shi and Hsu 2015]

Sen ominaisuuksiin kuuluu myös laaja suorituskyvyn dynaaminen skaalautuvuus, kuten virtualisoinnin yhteydessä puhuttiin. Tämän seurauksena onkin syntynyt useita erilaisia etärenderöintiä hyödyntäviä palveluita, joiden avulla voi esimerkiksi pelata graafisesti vaativia pelejä, joita pelatakseen joutuisi muutoin investoimaan suuriakin summia omaan tietokoneeseen.

Näistä palveluista tavallista kuluttajaa lähimpänä on Nvidia GRID. Se on Nvidian kehittämä näytönohjainten suorituskyvyn virtuaaliseen jakamiseen tarkoitettu palvelu. Siinä Nvidialta GRID-palvelun ostanut taho voi tarjota käyttäjilleen eritasoista graafista suorituskykyä heidän tarpeidensa mukaan. [Nvidia]

Yksi virtuaalisesti jaettu näytönohjain voi olla jaettuna jopa 16 käyttäjästä koostuvalle ryhmälle, jolloin tarjolla olevan suorituskyvyn määrä laskee, mutta samaan aikaan energiatehokkuus taas nousee. Toisaalta yhtä tällaista virtuaalisesti jaettua näytönohjainta voi käyttää vain yksikin henkilö, jolloin palvelu tarjoaa erittäin voimakasta suorituskykyä sitä tarvitseville. [Nvidia] Teknologian kehittyessä ja näytönohjainkohtaisen videomuistin määrän kasvaessa voi to-

dennäköisesti odottaa tämän maksimikäyttäjämäärän kasvavan entistä suuremmaksi.

5. Yhteenveto

Keskusyksiköiden energiatehokkuus on kehittynyt valtavasti viimeisen vuosikymmenen aikana. Vaikka tutkielmassa keskityttiinkin yksinkertaistetusti tarkastelemaan kehitystä pääasiassa keskusyksiköiden prosessorien näkökulmasta, ovat muutkin keskusyksiköissä käytetyt komponentit kehittyneet energiatehokkuutensa osalta. Keskusyksiköiden komponentteja on näin ollen kehitetty laskentateholtaan kyvykkäämmiksi ilman kompromisseja energiatehokkuuden parantamisessa. Samaan aikaan tavallisten toimisto-ohjelmien vaatima suorituskyky ei ole kuitenkaan juurikaan kasvanut, ja tämän seurauksena valmistajat ovat tarvinneet uusia keinoja tuotteidensa myynnin varmistamiseksi [Clark 2015]. Komponenttivalmistajat ovatkin alkaneet luoda erilaisia tuotelinjoja ja brändejä, joilla he pyrkivät profiloitumaan energiatehokkaina ja jopa ympäristöystävällisinä yhtiöinä [Intel].

Keskusyksiköiden markkinoilla on nähtävillä selkeä trendi kohti vähemmän kuluttavia ja silti hyvää laskentatehoa tarjoavia tuotteita. Näitä tuotteita varten valmistajat ovat kehittäneet paljon energiatehokkuutta parantavia tekniikoita, joista osa onkin omaksuttu lähes jokaisessa keskusyksikössä käytettäväksi. Osa näistä tekniikoista on menestynyt siksi, että käyttäjältä on otettu pois mahdollisuus vaikuttaa tekniikan toimintaan [Berl and Meer 2010].

Kuluttajien ja yritysten saatavilla on nykyään myös paljon tekniikoita, joiden avulla voidaan kehittää useiden keskusyksiköiden muodostamia kokonaisuuksia entistä energiatehokkaammiksi ja ympäristöä säästävämmiksi. Nämä virtualisointiin ja sen tuomiin mahdollisuuksiin perustuvat tekniikat mahdollistavat vanhojenkin keskusyksiköiden elinkaaren jatkamisen. Samalla ne pienentävät painetta tietotekniikan jatkuvaan uusimiseen noudattaen myös vihreän tietotekniikan periaatteita.

On mielenkiintoista nähdä, sovelletaanko näitä virtualisointiin pohjautuvia tekniikoita tulevaisuudessa yhä voimakkaammin myös tavallisiin keskusyksiköihin, ja muuttuuko ajan myötä keskusyksikön merkitys erilaiseksi. Vihreän talouskasvun mallin yleistyessä ja ihmisten kulutustottumusten muuttuessa entistä ympäristöystävällisemmiksi, keskusyksiköitä valmistavat yhtiöt pyrkivät todennäköisesti profiloimaan tuotteitaan entistä voimakkaammin vihreän tietotekniikan periaatteiden mukaan.

Viitteet

Andreas Berl and Hermann de Meer. 2010. A virtualized energy-efficient office environment. In: *Proceedings of the 1st International Conference on Energy-Efficient Computing and Networking*, 11-20.

Chao-Tung Yang, Kuan-Chieh Wang, Hsiang-Yao Cheng, Cheng-Ta Kuo and Ching-Hsien Hsu. 2011. Implementation of a green power management algorithm for virtual machines on cloud computing. In: *Ubiquitous Intelligence and Computing, Lecture Notes in Computer Science* 6905, 280-294.

Don Clark. 2015. Intel overhauls chips in bid to revive PC sales. http://www.wsj.com/articles/intel-overhauls-chips-in-bid-to-revive-pc-sales-1441155601 Checked 23.1.2016

Harold Castro, Mario Villamizar, German Sotelo, Cesar O. Diaz, Johnatan E. Pecero and Pascal Bouvry. 2012. Green flexible opportunistic computing with task consolidation and virtualization. *Cluster Computing* Sept. 2013, 16, 3, 545-557.

Intel. 2015. CPUs http://www.intel.com/content/www/us/en/homepage.html Checked: 12.12.2015

John Zysman and Mark Huberty. 2012. Green growth. *Intereconomics* 47, 3, 140-164.

Juanli Hu, Jiabin Deng and Juebo Wu. 2011. A green private cloud architecture with global collaboration. *International Journal of Precision Engineering and Manufacturing*, 13, 7, 1037-1045.

Karissa Miller and Mahmoud Pegah. 2007. Virtualization, virtually at the desktop. In: *Proceedings of the 35th Annual ACM SIGUCCS Fall Conference*, 255-260.

Karl R. Huppler. 2012. Performance per watt – benchmarking ways to get more for less. In: *Selected Topics in Performance Evaluation and Benchmarking, Lecture Notes in Computer Science* 7755, 60-74.

Lizhe Wang and Samee U. Khan. 2011. Review of performance metrics for green data centers: a taxonomy study. *The Journal of Supercomputing*, 63, 3, 639-656.

Nilton Bila, Eric J. Wright and Eyal de Lara, Kaustubh Joshi, H. Andrés Lagar-Cavilla, Eunbyung Park and Ashvin Goel, Matti Hiltunen and Mahadev Satyanarayanan. 2015. Energy-oriented partial desktop virtual machine migration. *ACM Transactions on Computer Systems*, 33, 1, Article 2.

Nilton Bila, Eyal de Lara, Kaustubh Joshi, H. Andrés Lagar-Cavilla, Matti Hiltunen and Mahadev Satyanarayanan. 2012. Jettison: Efficient idle desktop consolidation with partial VM migration. In: *Proceeding EuroSys '12 Proceedings of the 7th ACM European Conference on Computer Systems*, 211-224.

Nvidia. 2015. Nvidia GRID http://www.nvidia.com/object/grid-technology.html Checked: 30.12.2015

Passmark. 2015. CPU Benchmarks http://www.cpubenchmark.net/ Checked: 13.12.2015

Robert McMillan. 2015. PC sales continue to fall. http://blogs.wsj.com/digits/2015/07/09/pc-sales-continue-to-fall/ Checked 23.1.2016

Ryan Smith. 2011. AMD Radeon HD7970 review: 28nm and graphics core next, together as one. http://www.anandtech.com/show/5261/amd-radeon-hd-7970-review/11. Checked: 20.1.2016

Shu Shi and Cheng-Hsin Hsu. 2015. A survey of interactive remote rendering systems. *ACM Computing Surveys* 47, 4, Article 57.

Silvana Trimi and Sang-Hyun Park. 2012. Green IT: practices of leading firms and NGOs. *Service Business*, 7, 3, 363-379.

T. R. W. Scogland, H. Lin and W. Feng. 2010. A first look at integrated GPUs for green high-performance computing. In: *Computer Science - Research and Development*, Sept. 2010, 25, 3, 125-134.

Waler Binder and Niranjan Suri. 2009. Green computing: energy consumption optimized service hosting. In: Part 7 *SOFSEM 2009: Theory and Practice of Computer Science, Lecture Notes in Computer Science* 5404, 117-128.

Tabuetsintä

Anne Maunu

Tiivistelmä.
Tabuetsintä kuuluu metaheuristiikoiksi kutsuttuihin menetelmiin, jotka pyrkivät tarjoamaan tehokkaita ja tarpeeksi hyviin ratkaisuihin yltäviä menetelmiä kombinatoristen optimointiongelmien ratkaisemiseen. Se hyödyntää tässä prosessissa muistia, johon tallennetaan tietoa haun aikaista tapahtumista. Optimoinnista voidaan hyötyä useilla käytännön aloilla kuten logistiikassa. Tässä tutkielmassa esitellään tabuetsintää lähtien liikkeelle metaheuristiikkojen yleisestä merkityksestä optimoinnissa. Tämän jälkeen perehdytään tabuetsinnän perusteisiin. Lisäksi annetaan joitakin esimerkkejä menetelmän edistyneemmistä tekniikoista ja erilaisista sovellusalueista, joilla sitä on pyritty hyödyntämään.

Avainsanat ja -sanonnat: tabuetsintä, optimointi, metaheuristiikka

1. Johdanto

Tässä tutkielmassa tarkastellaan optimoinnissa hyödynnettävää *tabuetsintää* (tabu [sic] search). Tehokkaiden optimointimenetelmien kehittämisen tarve juontuu alkujaan käytännöllisistä lähtökohdista muun muassa telekommunikaation, logistiikan, kuljetuksen ja tuotannon aloilta [Glover and Laguna 1997]. Optimointiongelmien ratkaisemiseen on kehitetty useita erilaisia menetelmiä. Kompleksisuusteorian tutkiminen 1970-luvulla osoitti, että osa näistä ongelmista on niin vaikeita, ettei niiden kaikkien ratkaisuvaihtoehtojen tutkiminen ja täten myöskään täysin optimaalisen ratkaisun etsiminen ole realistista. Syntyi siis tarve luoda erilaisia menetelmiä, joiden avulla ongelmille voitaisiin hakea tarpeeksi hyviä ratkaisuja tehokkaasti. Tähän tarkoitukseen on syntynyt useita erilaisia *metaheuristiikkoja* (metaheuristics), joita laajalti käytetään kombinatoristen optimointiongelmien ratkaisemisessa. Fred Glover esitteli tabuetsinnän ensimmäistä kertaa vuonna 1986. Menetelmästä on sittemmin tullut suosittu kombinatoristen ongelmien ratkaisemisessa ja aiheesta on viimeisten 25 vuoden aikana tehty lukuisia tutkimuksia. [Gendreau and Potvin 2010]

Tutkielman tavoitteena on luoda katsaus tabuetsintään ja kontekstiin, jossa sitä sovelletaan. Toisessa luvussa tarkastellaan yleisesti metaheuristiikkoja ja niiden periaatteita kombinatorisessa optimoinnissa. Kolmannessa luvussa esitellään tabuetsinnän periaatteita ja käsitellään havainnollistavaa esimerkkiä kyseisen metaheuristiikan toiminnasta. Neljäs luku keskittyy kuvailemaan joitakin edistyneempiä tekniikoita, jotka hyödyntävät tabuetsinnän ideoita. Hyb-

ridisoinnin yhteydessä esitellään lyhyesti myös muita yleisesti käytettyjä metaheuristiikkoja. Viidennessä luvussa esitellään erilaisia ongelmia ja sovellusalueita, joissa tabuetsintä on osoittautunut hyödylliseksi. Kuudes luku päättää tutkielman yhteenvetoon.

2. Metaheuristiikat optimoinnissa

Optimoinnissa pyritään löytämään jollekin ongelmalle kaikkien mahdollisten ratkaisujen joukosta ”paras” ratkaisu [Blum and Roli 2003]. Tässä yhteydessä parhaus määritellään tapauskohtaisesti jollain sopivalla kriteerillä. *Kombinatorisessa optimoinnissa* (combinatorial optimization) sallittujen ratkaisujen määrä on äärellinen. Ne kaikki voitaisiin siis käydä läpi, mutta se vaatisi hyvin paljon laskenta-aikaa. Tästä syystä monesti hyödynnetään approksimointialgoritmeja, jotka eivät käsittele kaikkia mahdollisia ratkaisuja, mutta pyrkivät löytämään tarpeeksi hyviä ratkaisuja nopeasti. Menetelmiä on kahdenlaisia: ratkaisu voidaan koota lisäämällä siihen osia, kunnes ratkaisusta tulee kokonainen, tai haku voi alkaa jostain alustavasta ratkaisusta, josta pyritään iteratiivisesti siirtymään parempaan ratkaisuun. [Blum and Roli 2003]

Metaheuristiikat ovat yksi approksimointialgoritmien laji. Siinä missä perinteiset heuristiikat keskittyvät tietynlaisten ongelmien ratkaisemiseen, metaheuristiikat ovat yleisiä menetelmiä, joita voidaan soveltaa monenlaisiin optimointiongelmiin [Talbi 2009]. Niiden avulla hakuavaruuksia voidaan tutkia tehokkaasti. Metaheuristiikoille olennaista on pyrkimys tasapainon ylläpitämiseen *hajauttamisen* (diversification) ja *tehostamisen* (intensification) välillä. Itse asiassa nämä termit pohjautuvat juuri tabuetsintään. [Blum and Roli 2003] Tehostamisen tarkoituksena on keskittyä tutkimaan hakuavaruuden sellaisia osia, joissa ratkaisut vaikuttavat lupaavilta. Hajauttamisella sen sijaan pyritään ohjaamaan etsintä myös muualle hakuavaruuteen, jottei haku keskittyisi liian pieneen osaan siitä, sillä hyvälaatuisia ratkaisuja voi löytyä muualtakin. Muutoin saatetaan jäädä kauaskin optimaalisesta ratkaisusta. [Gendreau and Potvin 2010] Englanninkielisessä kirjallisuudessa käytetään joskus myös vaihtoehtoisia termejä exploration ja exploitation, mutta ne liittyvät lähinnä satunnaisuuteen perustuviin lyhyen aikavälin strategioihin [Blum and Roli 2003].

Metaheuristiikkoja voidaan luokitella useilla eri tavoilla. Tällaista jaottelua voidaan tehdä esimerkiksi sen perusteella, onko metaheuristiikan luomisessa käytetty inspiraationa jotain luonnossa esiintyvää ilmiötä. Toinen luokittelutapa pohjautuu siihen, käsitteleekö algoritmi kerrallaan vain yhtä ratkaisua vai suurempaa ratkaisupopulaatiota. [Blum and Roli 2003] Tämä jaottelu lienee luvussa 4 käsiteltävän metaheuristiikkojen hybridisoinnin kannalta relevantein,

sillä kyseistä luokittelua voidaan käyttää perustana eri metaheuristiikkojen yhdistelemiselle [Blum and Roli 2003].

3. Tabuetsinnän peruspiirteitä

Tabuetsintä on yksi metaheuristiikoista. Optimointiongelmissa pyritään optimoimaan, eli minimoimaan tai maksimoimaan, tavoitefunktio $f(x)$, kun $x \in X$, missä X voidaan käsittää ratkaisujoukkona eli hakuavaruutena, jossa on tiettyjä rajoituksia ratkaisujen muodostamiselle. Tabuetsintä hyödyntää *lokaalin haun* (local search) periaatteita lähtemällä liikkeelle jostain ratkaisusta ja edeten iteratiivisesti ratkaisusta toiseen. Kuhunkin ratkaisuun $x \in X$ liittyy naapurusto $N(x)$, jonka ratkaisut $x' \in N(x)$ ovat saavutettavissa ratkaisusta x siirroksi kutsutun operaation avulla. Yksinkertaisessa laskeutumismenetelmässä pyritään minimoimaan funktio $f(x)$, jolloin hyväksytään siirtyminen vain edellistä parempaan ratkaisuun ja haku päätyy, kun käsiteltävällä ratkaisulla ei ole enää sitä parempia naapuriratkaisuja. Menetelmän huonona puolena on se, ettei löydetty lokaali minimi useinkaan ole globaali minimi, eli $f(x)$ ei minimoidu kaikkien $x \in X$ suhteen. [Glover and Laguna 1997]

Tabuetsintää voidaan jatkaa myös lokaalin optimin saavuttamisen jälkeen, sillä menetelmä hyväksyy myös siirrot, jotka eivät paranna ratkaisua. Tällöin on estettävä haun palaaminen takaisin samaan lokaaliin optimiin ja yleisemmin päätyminen kiertämään kehää samoissa ratkaisuissa. Käytännössä tämä tapahtuu merkitsemällä jo käydyt ratkaisut tabuiksi, ettei äskettäin käsiteltyihin ratkaisuihin palattaisi uudelleen. [Glover 1989] Siirtoja tehdessä tutkitaankin koko naapuruston $N(x)$ sijasta muokattua naapurustoa $N^*(x)$, joka yksinkertaisessa lyhytaikaista muistia hyödyntävässä tabuetsinnässä sisältää kaikki ne naapuriratkaisut, joita ei ole merkitty tabuiksi. Pidempiaikaista muistia hyödynnettäessä on mahdollista sisällyttää kyseiseen joukkoon muitakin ratkaisuja. Jos tabuiksi luokitellut ratkaisut on merkitty tabulistalle T, pätee siis $N^*(x) = N(x) \setminus T$. [Glover and Laguna 1997] Seuraavaksi käsiteltäväksi ratkaisuksi valitaan kullakin iteraatiolla tästä uudesta naapurustosta $N^*(x)$ suurimman parannuksen tavoitefunktion arvoon tuottava ratkaisu, tai jos sellaista ei ole, pienimmän arvon huononemisen aiheuttava [Glover 1989].

Algoritmissa 1 on esitetty hyvin yksinkertainen ja karkea versio tabuetsinnästä minimointitehtävän yhteydessä. Siinä käytetään samoja merkintöjä kuin aikaisemmin esiteltiin ja lisäksi parhaasta toistaiseksi löytyneestä ratkaisusta käytetään merkintää x^*. Haku jatkuu siihen asti, kunnes jokin lopetuskriteeri täyttyy tai joukko $N^*(x)$ tyhjenee, jolloin saavutettavissa ei luonnollisestikaan enää ole naapuriratkaisuja, joihin siirtyä. Tabulistan T päivittämiseen kuuluu siirron lisääminen listalle sekä mahdollisesti jonkin aikaisemman (tabulistalla

kauimmin olleen) tabun poistaminen. Algoritmia voidaan toistaa useaankin kertaan, jos liikkeelle lähdetään eri ratkaisuista.

---

valitse jokin alustava ratkaisu $x \in X$
$x^* \leftarrow x$
$T \leftarrow \varnothing$
while lopetuskriteeri ei täyttynyt
    valitse paras ratkaisu $x' \in N^*(x)$
    $x \leftarrow x'$
    if $f(x) < f(x^*)$
        $x^* \leftarrow x$
    päivitä T

Algoritmi 1. Tabuetsintäalgoritmin yksinkertaistettu muoto.

Usean tabulistan käyttäminen yhtä aikaa on mahdollista ja usein kannattavaakin. Esimerkiksi useanlaisten siirtojen käyttäminen naapuruston muodostamisessa voi synnyttää tarpeen käyttää tabulistoja kullekin erityyppiselle siirtomuodolle. [Gendreau and Potvin 2010] Tabulistan toteutuksessa kannattaakin ottaa huomioon ratkaistavan ongelman konteksti. Joissain yhteyksissä voi olla tehokasta liittää ratkaisun ominaisuuksiin iteraationumero, joka ilmaisee, millä iteraatiolla kyseinen ominaisuus on merkitty tabuksi tai vastaavasti milloin tabustatus raukeaa. [Glover and Laguna 1997] Tässä tutkielmassa viitataan tabulistan koolla sekä varsinaisen tabulistan pituuteen, kun tabuja ratkaisuja tai ominaisuuksia kirjataan erilliselle tabulistalle, että tähän vaihtoehtoiseen tabun kestoon perustuvaan tapaan.

Usein tabulistojen koko pysyy muuttumattomana haun aikana. On mahdollista hyödyntää myös vaihtelevan mittaisia tabulistoja ja onkin osoittautunut, ettei kiinteän kokoinen tabulista aina onnistu estämään kiertämistä kehää samoissa ratkaisuissa. [Gendreau and Potvin 2010] Tehokas koko tabulistalle riippuu yleisesti ratkaistavan ongelman koosta. Mitä rajoittavampia tabut ovat, sitä lyhyempi lista yleensä riittää. Sopiva koko löytyykin usein kokeilemalla: toistuvat ratkaisut kertovat liian lyhyistä tabulistoista ja ratkaisujen laadun heikkeneminen liian pitkistä. Tabulistan koon muuttamisella haun aikana voi olla hyödyllisiä seurauksia. Lyhyillä tabulistoilla voidaan tutkia tarkasti lokaalin optimin ympäristöä ja näin tehostaa hakua. Pidemmät listat taas ajavat hakua pois lokaalista optimista, jolloin haku hajaantuu. Satunnainen koon vaihtelu jollain tietyllä vaihteluvälillä on melko vaivatonta toteuttaa, mutta koon sys-

temaattinen muuttaminen tuottaa usein parempia tuloksia. [Glover and Laguna 1997]

Tabulistaan ei yleensä kannata kirjata kokonaisia ratkaisuja vaan joitakin niihin liittyviä attribuutteja, esimerkiksi ratkaisuihin liittyviä komponentteja, siirtoja tai eroja kahden ratkaisun välillä. Tämä tosin johtaa siihen, että käsittelemätönkin ratkaisu saatetaan tulkita tabuksi. Jotta hyvälaatuiset ratkaisut eivät tästä syystä jäisi löytymättä, käytetään *aspiraatiokriteerejä* (aspiration criteria). [Blum and Roli 2003] Ne mahdollistavat siirtymisen myös tabuiksi luokiteltuihin ratkaisuihin. Näiden kriteerien onnistunut hyödyntäminen saattaa vaikuttaa huomattavasti tabuetsinnän tehokkuuteen. Yhdessä usein käytetyssä yksinkertaisessa aspiraatiokriteerissä ratkaisun tabustatus kumotaan, mikäli se tuottaa paremman tuloksen kuin toistaiseksi paras löydetty ratkaisu. On kuitenkin olemassa muitakin aspiraatiokriteerejä, joiden käyttäminen voi parantaa hakua. Jos kaikki saavutettavissa olevat ratkaisut on merkitty tabuiksi, voidaan valita siirryttäväksi niin sanotusti vähiten tabuun naapuriratkaisuun. Tämä ratkaisu voi olla vaikkapa se, jonka tabustatus on poistumassa ensin. Jos haun suunta ei ole muuttunut hetkeen, eli ratkaisut eivät ole erityisesti parantuneet eivätkä huonontuneet, voi myös olla hyödyllistä tehdä tabuiksi luokiteltuja siirtoja. [Glover and Laguna 1997]

Aspiraatiokriteereissä on lisäksi hyvä huomioida siirtojen *vaikutus* (influence). Se liittyy siirtojen aiheuttamiin muutoksiin esimerkiksi ratkaisun rakenteen kannalta. Suuri vaikutus ei välttämättä tarkoita, että ratkaisu välittömästi paranisi, mutta suuren vaikutuksen siirrot ovat hyödyllisiä, jotta voidaan estää haun keskittyminen vain lokaalin optimin ympärille. Aspiraatiokriteeri voidaan rakentaa myös tämän tiedon varaan ja sallia tabuksi luokiteltu pienen vaikutuksen siirto, mikäli jollain sen tabustatuksen muuttumisen jälkeen tehdyllä siirrolla on ollut suuri vaikutus. [Glover and Laguna 1997]

Haku päättyy sille määritellyn lopetuskriteerin täytyttyä. Tällaisen kriteerin käyttäminen on välttämätöntä, koska muuten teoriassa etsintä saattaisi jatkua ikuisesti. Etsintä voidaan lopettaa esimerkiksi silloin, kun on käyty tietty määrä iteraatioita joko haun alkamisen tai viimeisimmän paremman ratkaisun löytymisen jälkeen. [Gendreau and Potvin 2010]

3.1. Esimerkki

Havainnollistetaan seuraavaksi tabuetsintää pienellä esimerkillä. Valitaan tehtäväksi etsiä minimivirittävä puu kuvan 1 suuntaamattomasta painotetusta graafista. Tavallisesti tällaisen tehtävän ratkaiseminen voitaisiin hoitaa jonkin suoraviivaisen *ahneen algoritmin* (greedy algorithm) avulla [Glover 1990], joten tabuetsinnän käyttäminen tuskin edes olisi erityisen hyödyllistä. Määritellään-

kin siis puun muodostamiselle pari sopivaa rajoitusta, jotta tabuetsinnän piirteet näkyvät tässä esimerkissä. Käytetään merkintää

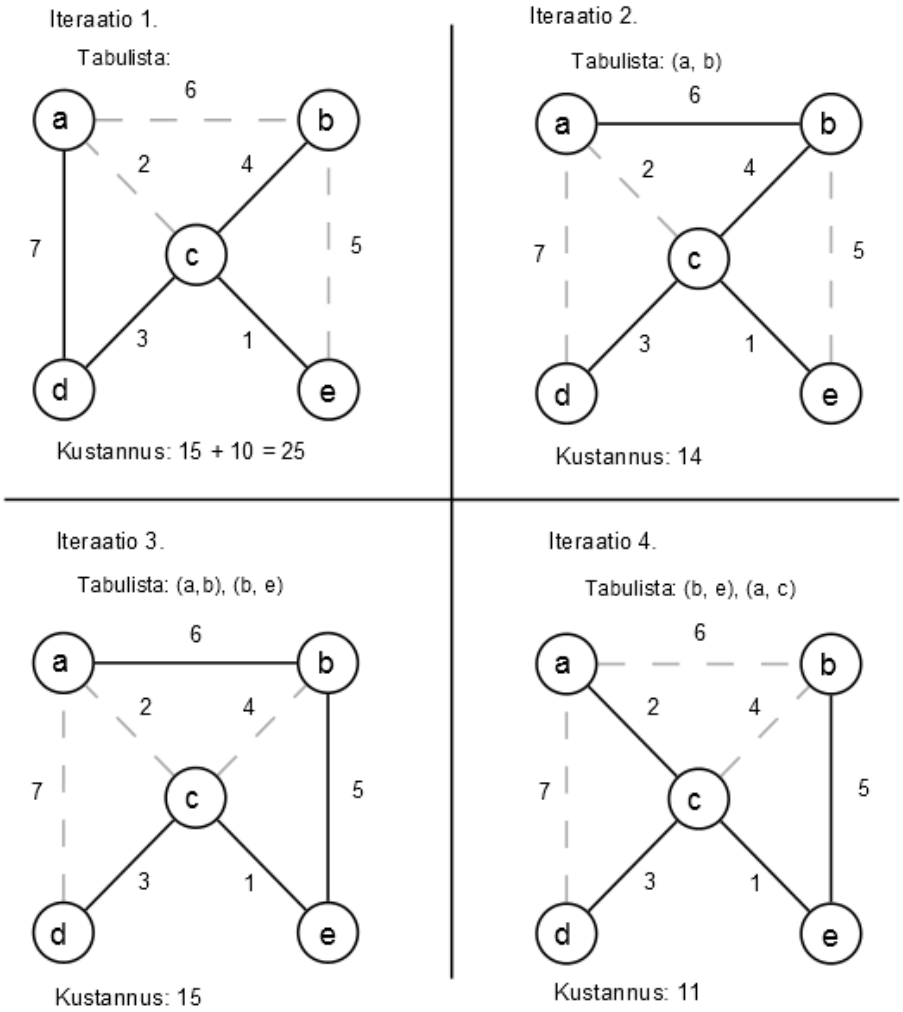
$$(x,\ y) = \begin{cases} 1, & \text{kun kaari (x, y) kuuluu puuhun,} \\ 0, & \text{kun kaari (x, y) ei kuulupuuhun.} \end{cases}$$

Nyt rajoitukset voidaan esittää muodossa

(a, c) ≤ (b, e) ja

(a, d) + (b, c) + (b, e) ≤ 1.

Ensimmäinen rajoituksen mukaan kaari (a, b) ei saa esiintyä puussa, jos kaari (b, e) ei kuulu siihen. Toinen rajoitus sen sijaan kertoo, että kaarista (a, d), (b, c) ja (b, e) vain yksi saa kerrallaan kuulua puuhun. Mahdollistetaan näiden rajoitusten rikkominen, mutta sovitaan jokaisen rikkeen kasvattavan puun kustannusta 10:llä.



Kuva 1. Minimivirittävän puun etsimistä tietyin rajoituksin graafista.

Graafissa on yhteensä viisi solmua, joten minimivirittävän puun muodostamiseen tarvitaan neljä kaarta. Kuvassa kulloinkin käsiteltävä puu on kuvattu mustilla yhtenäisillä viivoilla ja muut mahdolliset kaaret harmain katkoviivoin. Kaarien vieressä olevat numerot kuvaavat niiden kustannuksia, joiden avulla voidaan laskea koko puun kustannus mahdolliset rajoitusten rikkomukset huomioiden. Siirrot ratkaisusta toiseen tapahtuvat lisäämällä yksi kaari puuhun poistaen samalla yksi. Kaaren lisääminen aiheuttaa syklin syntymisen, joten puurakenteen ylläpitämiseksi poistettava kaari on aina jokin tähän sykliin kuuluvista.

Valitaan tabulista siten, että se sisältää kerrallaan korkeintaan kaksi tabua, joista vanhempi poistetaan aina lisättäessä uusi listan ollessa täynnä. Merkitään jokaisen siirron yhteydessä tabuksi lisätty kaari. Nyt kyseistä kaarta ei voida heti poistaa puusta, mikä saattaisi johtaa siirtymisen takaisin edelliseen ratkaisuun. Hyödynnetään aspiraatiokriteerejä, jos ajaudutaan tilanteeseen, jossa muuten jouduttaisiin siirtymään huonompaan ratkaisuun. Mikäli jokin tabuksi merkitty ratkaisu on tässä tilanteessa parempi kuin toistaiseksi paras löydetty ratkaisu, kumotaan kaaren tabustatus.

Muodostetaan ensimmäiselle iteraatiolle alkutilanne, jossa puun kustannus on 25. Kaarien kustannusten summan lisäksi siihen kuuluu rajoitusten rikkomisesta aiheutuva sakko, koska sekä kaari (a, d) että (b, c) kuuluvat puuhun. Taulukossa 1 on tarkasteltu ensimmäisen iteraation naapuriratkaisujen kustannuksia. Ilman asetettuja rajoituksia olisi kannattavaa poistaa kaari (a, d) ja lisätä sen tilalle kaari (a, c). Tämä olisikin siinä tapauksessa paras ratkaisu koko tehtävälle. Rajoitukset huomioiden paras ratkaisu on sen sijaan selvästikin poistaa puusta kaari (a, d) ja lisätä siihen kaari (a, b). Tämä ratkaisu ei riko asetettuja rajoituksia, ja kustannus laskee 14:ään.

Lisää	**Poista**	**Kustannus**
(a, b)	(a, d)	14
(a, b)	(b, c)	17
(a, b)	(c, d)	18 + 10 = 28
(a, c)	(a, d)	10 + 10 = 20
(a, c)	(c, d)	14 + 10 = 24
(b, e)	(b, c)	16 + 10 = 26
(b, e)	(c, e)	19 + 10 = 29

Taulukko 1. Esimerkin ensimmäisen iteraation siirtovaihtoehdot.

Toisen iteraation ratkaisu on parempi kuin ensimmäinen, joten se on tällä hetkellä paras löydetty. Tabulistalla on kaari (a, b), joka lisättiin toisen iteraation ratkaisun saavuttamiseksi. Nyt ei löydetä enää siirtoa, joka suoraan parantaisi ratkaisua. On siis päädytty lokaaliin minimiin. Tabulistalla on vain yksi kaari, jonka tabustatusta ei kannata kumota, sillä kustannus nousisi taas suureksi. Vähiten huono vaihtoehto onkin lisätä seuraavaksi kaari (b, e) ja poistaa samalla kaari (b, c). Kustannus nousee yhdellä, mutta tabuetsinnän periaatteen mukaan päästään siirtymään pois lokaalista minimistä. Tabulistaan lisätään nyt toinenkin kaari, (b, e).

Myöskään kolmannella iteraatiolla ei heti löydetä parempaa ratkaisua kuin se, jossa parhaillaan ollaan. Kaikki sallitut naapuriratkaisut ovat kustannuksiltaan suurempia. Onkin aika hyödyntää aikaisemmin asetettua aspiraatiokriteeriä. Tabulistalle ensin lisätyn kaaren poistaminen puusta mahdollistaa sellaisen ratkaisun muodostamisen, jossa kustannukseksi tulee 11. Se on siis parempi ratkaisu kuin toistaiseksi paras löydetty. Kriteerin mukaan kyseisen kaaren tabustatus voidaan siis kumota. Nyt puusta poistetaan tämä kaari (a, b) ja siihen lisätään kaari (a, c), josta ei aiheudu lisäkustannuksia, koska myös kaari (b, e) kuuluu rajoitusten mukaisesti puuhun. Hakua olisi mahdollista vielä jatkaa, mutta koska nyt löydetty ratkaisu on myös itse asiassa globaali minimi, voidaan esimerkkihaku päättää iteraatioon 4.

3.2. Muistin hyödyntäminen tabuetsinnässä

Tabuetsinnässä muistin hyödyntämisellä on tärkeä rooli. Esimerkiksi tabulistan käyttö perustuu pitkälti lyhytaikaiseen muistiin. Myös pidempiaikaista muistia on mahdollista hyödyntää haussa. Vaikka yksinkertaiset lyhytaikaiseen muistiin nojautuvat lähestymistavat ovat joskus yllättävänkin menestyksekkäitä, pidempiaikaisen muistin onnistunut hyödyntäminen usein tehostaa hakumenetelmiä. [Glover and Laguna 1997]

Tabuetsinnän muistirakenteet soveltavat neljää periaatetta: *viimeaikaisuutta* (recency), *toistumistiheyttä* (frequency), *laatua* (quality) ja vaikutusta [Glover and Laguna 1997]. Näistä viimeinen nousi esiin jo aspiraatiokriteerien yhteydessä. Se liittyy haun aikana tehtyjen valintojen vaikutukseen niin ratkaisun laadun kuin rakenteenkin kannalta. Laadulla sen sijaan voidaan eritellä haun aikana löydettyjen ratkaisujen arvoa. Tällaisen muistin avulla voidaan tunnistaa hyviin ratkaisuihin liittyviä komponentteja sekä tällaisiin ratkaisuihin johtavia polkuja. Näin pystytään ohjailemaan hakua kohti hyviä ratkaisuja ja pyrkiä välttämään huonoja. [Glover and Laguna 1997]

Viimeaikaisuus ja toistumistiheys täydentävät toisiaan. Tabuetsinnässä yleisin käytetty muistiperiaate on viimeaikaisuus, sillä se näkyy suoraan tabulistan

hyödyntämisessä. Toistumistiheys täydentää viimeaikaisuuden tarjoamaa informaatiota ja tukee hyvien siirtojen valitsemista. Sen avulla voidaan pitää kirjaa siitä, monellako iteraatiolla ratkaisujen yksittäiset ominaisuudet muuttuvat, eli milloin ne lisätään ratkaisuun tai poistetaan siitä, sekä monellako iteraatiolla ominaisuudet kuuluvat käytyihin ratkaisuihin. Näiden tietojen avulla voidaan selvittää, kuinka usein ominaisuudet muuttuvat ja kuinka usein ominaisuudet ovat ratkaisujen osina. Tästä taasen voidaan päätellä, milloin hakua on tarpeen hajauttaa. Jos ominaisuus on usein osana hyviä ratkaisuja, sitä voidaan pitää houkuttelevana, mutta sitä vastoin huonoissa ratkaisuissa monesti esiintyvät ominaisuudet tuskin ovat erityisen hyviä. Usein muuttuvat ominaisuudet liittyvät yleensä ratkaisun hienosäätöön. Toistumistiheyteen perustuva muisti on hyödyksi, kun pyritään tunnistamaan hakuavaruudesta houkuttelevia siirtoja, jotta niiden tekemistä voidaan suosia ja samalla vältellä mahdollisesti huonoja siirtoja. Huonoihin siirtoihin voi tällöin liittyä rangaistuksia, jolloin toistumistiheys ja viimeaikaisuus kumpikin vaikuttavat siirtojen valitsemiseen. [Glover and Laguna 1997]

Tabuetsinnässä pitkäaikaiseen muistiin voidaan tallentaa kokonaisia ratkaisuja tai niihin liittyviä ominaisuuksia. Kokonaisia tallennettavia ratkaisuja voivat olla haun aikana vieraillut erityisen hyvät eliittiratkaisut tai niiden vielä vierailemattomat lupaavilta vaikuttavat naapuriratkaisut. Ratkaisuihin liittyviä muistissa pidettäviä ominaisuuksia voidaan hyödyntää hakua ohjailevasti. Tällaisia ominaisuuksia ovat esimerkiksi graafeissa ratkaisuihin lisätyt tai niistä poistetut kaaret tai solmut. [Glover and Laguna 1997]

4. Edistyneitä tekniikoita

Edellinen luku käsitteli pääasiassa tabuetsinnän perusajatuksia. Jo niitä soveltamalla on mahdollista saada aikaan onnistuneita tuloksia, mutta lisäksi on kehitelty yhä tehokkaampia menetelmiä tabuetsinnän hyödyntämiseen. Tarkastellaan seuraavaksi joitakin näistä konsepteista.

4.1. Strateginen oskillointi

Strateginen oskillointi (strategic oscillation) pyrkii hyödyntämään tehokkaasti tehostamisen ja hajauttamisen välistä vuorovaikutusta. Menetelmän keskeisenä ajatuksena on oskillointiraja, joka kuvaa käsiteltävälle ongelmalle tyypillistä kriittistä tasoa, joka voi määräytyä esimerkiksi ratkaisun kelvollisuuden perusteella. Usein raja kuvastaa kohtaa, johon haku normaalisti pysähtyisi. Strategisessa oskilloinnissa voidaan jatkaa eteenpäin tästä pisteestä hyväksymällä rajan ylittäminen. Rajan yli voidaan jatkaa jonkin ennalta määritellyn syvyyden verran, minkä jälkeen palataan takaisin kohti oskillointirajaa ja uudelleen sen yli

vastakkaisesta suunnasta. Menetelmän oskilloiva piirre syntyy, kun rajaa lähestytään vuorotellen kummastakin suunnasta. Samalla tabuetsinnän perusmekanismit estävät samojen ratkaisujen toistumisen. [Glover and Laguna 1997]

Strategisen oskilloinnin hyödyntämistapa riippuu jossain määrin ratkaistavan ongelman ominaisuuksista. Graafiteoriassa oskillointiraja kuvaa toivottuja ominaisuuksia graafin rakenteessa. Rakenteeseen voidaan vaikuttaa lisäämällä siihen kaaria, solmuja tai aligraafeja. Oskillointirajana voitaisiin esimerkiksi pitää kohtaa, jossa metsään muodostuu syklejä. Tällöin joudutaan kenties muuttamaan siirtojen taustalla olevia sääntöjä. Takaisin kohti oskillointirajaa käännyttäessä graafin osien lisäykset vaihtuvat niiden poistoiksi. [Glover and Laguna 1997]

Jos ratkaistavana ongelmana olisi luoda graafista aligraafi, johon kuuluu tietty määrä kaaria ja jonka yhteispaino on mahdollisimman pieni, olisi oskillointiraja tämä haluttu kaarien määrä. Kaaria voitaisiin kuitenkin lisätä enemmänkin. Kaarien poistamiseen täytyy tällöin soveltaa jotain erilaista sääntöä kuin niiden lisäämiseen, jottei päädyttäisi vain peruuttamaan tehtyjä lisäyksiä. Tällaisessa tehtävässä kelvollisia ratkaisuja ovat pelkästään oskillointirajalla olevat ratkaisut, sillä muu määrä kaaria aligraafissa ei ole ratkaisussa sallittu. Voikin kenties olla hyödyllistä liikkua vain toispuoleisesti, tässä tapauksessa rajan yläpuolella, jolloin kaaria ei koskaan ole sallittua määrää vähemmän. [Glover and Laguna 1997]

Välillä on myös kannattavaa pysytellä hyvin lähellä oskillointirajaa ja pitää oskilloinnin syvyys pienenä, jotta hakuavaruuden tutkiminen on tehokasta. Yksi vaihtoehto on pysäyttää oskillointi hetkeksi ja hakeutua lokaaliin optimiin aina saavutettaessa raja. Tehostamisessa voidaan myös suosia joitain ratkaisujen ominaisuuksia tekemällä niiden poistamisesta vaikeampaa. Oskillointi voi hyödyntää myös pitkäaikaista muistia haun hajauttamiseen. [Glover and Laguna 1997]

4.2. Probabilistinen tabuetsintä ja kandidaattilistastrategiat

Perinteinen tabuetsintä huomioi yleensä kaikki elementit ratkaisun naapurustossa $N(x)$. Se saattaa vaikuttaa negatiivisesti haun tehokkuuteen. Yksi vaihtoehto on tarkastella satunnaisesti koostettua naapuriratkaisujen ryhmää $N'(x)$, joka sisältää vain osan naapurustosta $N(x)$. Lisähyötynä tällaisesta satunnaistamisesta on siinä, että sekin ehkäisee kehän kiertämistä samoissa ratkaisuissa, jolloin tabulistan kokoa voidaan pienentää. Toisaalta menettely voi jopa johtaa siihen, ettei kaikkia hyviä ratkaisuja löydetä, jos jokin lokaali optimi tulee vahingossa sivuutettua. [Gendreau and Potvin 2010]

Toinen, strategisempi tapa vähentää tarkasteltavien naapuriratkaisujen määrää on hyödyntää kandidaattilistoja [Gendreau and Potvin 2010]. Siirtymissä ratkaisusta seuraavaan kandidaattiratkaisujen generointi ja arviointi voivat osoittautua kriittisiksi haun kannalta. Kandidaattilistoille voidaan sisällyttää sellaisia siirtoja naapurustosta $N^*(x)$, joilla on tiettyjä haluttuja ominaisuuksia. Kandidaattilistastrategiat voivat perustua joihinkin yleisiin strategioihin tai kontekstiriippuvaisiin sääntöihin. Jälkimmäisissä periaatteena voi olla esimerkiksi kynnysarvojen määritteleminen siirtojen laadulle, eliittikandidaattien listaaminen käsiteltäviksi tai siirtoihin kuuluvien osaoperaatioiden tarkasteleminen. Kandidaattilistastrategioiden tehokkuuden arvioinnissa ei niinkään kannata kiinnittää huomiota yksittäisiin iteraatioihin kuluvaan laskentaan vaan pikemminkin parhaan löydetyn ratkaisun laatuun tietyn laskennallisen ajan puitteissa. [Glover and Laguna 1997]

Probabilistinen tabuetsintä (probabilistic tabu search) pyrkii hyödyntämään satunnaistamista [Gendreau and Potvin 2005]. Siinä arvioidaan ensin mahdollisia siirtoja niiden tabustatuksen ja muiden tabuetsinnälle relevanttien ominaisuuksien kannalta. Seuraavaksi nämä arviot kuvataan positiiviksi painoarvoiksi, jotka jaetaan painojen summalla todennäköisyyksiksi. Näin voidaan suosia parhaiksi arvioitujen siirtojen valitsemista. Kandidaattilistastrategioilla on merkittävä rooli probabilistisessa tabuetsinnässä, sillä niiden avulla voidaan valita sopivia siirtoja arvioitaviksi. Probabilistisen lähestymistavan etuna on se, että sillä voidaan tasoittaa tilannetta, jossa paras arviointi ei välttämättä vastaa parasta siirtoa. [Glover and Laguna 1997] On toki mahdollista myös yhdistää elementtejä probabilistisesta ja ei-probabilistisesta tabuetsinnästä. Lisäksi ideaa voidaan hyödyntää myös pidempiaikaisen muistin yhteydessä. [Glover 1989]

4.3. Reaktiivinen tabuetsintä

Jotkin tabuetsinnän sovellukset kieltävät siirtymisen kaikkiin sellaisiin ratkaisuihin, joita on jo käsitelty. Menettely saattaa kuitenkin osoittautua hitaaksi ja jopa estää optimaalisen ratkaisun löytymisen, mikäli hyvät ratkaisut keskittyvät liian lähelle toisiaan. Usein käytetäänkin jotain tietyn mittaista tabulistaa, jolloin listan koon valitseminen oikein on hyvin tärkeää, sillä muuten haku saattaa juuttua kiertämään kehää. Jotkin menetelmät pyrkivät välttymään näiltä ongelmilta käyttämällä tabulistaa, jonka koko vaihtelee satunnaisesti. [Battiti and Tecchiolli 1994]

Reaktiivinen tabuetsintä (reactive tabu search) hyödyntää myös muuttuvaa tabulistan pituutta, mutta pituus mukautuu kulloinkin käsillä olevaan ongelmaan ja haun kehittymiseen. Käsitellyistä ratkaisuista pidetään kirjaa, jotta haun aikana voidaan valvoa ratkaisujen toistumista ja näiden toistumien välis-

ten iteraatioiden määrää. Jotta ratkaisujen toistumista pystytään tehokkaasti seuraamaan, voidaan käyttää esimerkiksi *hajautusta* (hashing), jottei kokonaisia käytyjä ratkaisuja tarvitse kirjata muistiin. Mikäli toisteisuutta esiintyy liikaa, voidaan tabulistan pituutta kasvattaa nopeasti. Tabulistan pituutta voidaan vastaavasti pienentää, kun ajaudutaan sellaisille alueille hakuavaruudessa, joissa pitkästä listasta ei enää ole hyötyä. Myös pitkäaikaista muistia voidaan hyödyntää ja sen perusteella hajauttaa hakua, kun haun todetaan keskittyvän liian pieniin alueisiin hakuavaruudessa. [Battiti and Tecchiolli 1994]

4.4. Hybridisointi

Metaheuristiikkojen hybridisoimista on tutkittu useista eri lähtökohdista. Metaheuristiikkojen yhdisteleminen keskenään on osoittautunut tehokkaaksi menetelmäksi kombinatoristen optimointiongelmien ratkaisemisessa. [Talbi 2002] Yksi suosittu hybridisointitapa on ollut populaatioperustaisten ja kerrallaan yhtä ratkaisua käsittelevien metaheuristiikkojen yhdistäminen. Tällaisilla menetelmillä on saatu aikaan hyviä tuloksia, eikä ihmekään, sillä ensin mainitut ovat hyviä lupaavalta vaikuttavien alueiden tunnistamisessa hakuavaruudesta ja jälkimmäiset sen sijaan soveltuvat hyvin näiden lupaavien alueiden tarkempaan tutkimiseen. [Blum and Roli 2003] On olemassa muitakin tapoja hybridisoida metaheuristiikkoja kuin yhdistellä niitä muiden metaheuristiikkojen kanssa. Esimerkiksi Blum ja muut [2011] erittelevät lisäksi hybridisointia *rajoiteohjelmoinnin* (constraint programming), puuhakujen, *ongelmien relaksoinnin* (problem relaxation) ja dynaamisen optimoinnin kanssa. Keskitytään seuraavaksi kuitenkin lähinnä esittelemään joitakin metaheuristiikkoja, joihin tabuetsintää on onnistuneesti yhdistelty.

4.4.1. Simuloitu jäähdytys

Simuloitu jäähdytys (simulated annealing) on yksi vanhimmista metaheuristiikoista. Kuten tabuetsinnässä, myös simuloidussa jäähdytyksessä hyväksytään siirtyminen käsiteltävästä ratkaisusta huonompaan ratkaisuun. Haussa hyödynnetään lämpötilaparametria, joka vaikuttaa siihen, millaisella todennäköisyydellä huonompi ratkaisu hyväksytään. Haun alussa lämpötilaparametrin arvo on suuri, jolloin myös huonon ratkaisun hyväksyminen on todennäköisempää. Haun edetessä arvoa kuitenkin lasketaan, jolloin valinta kohdistuu yhä enenevissä määrin parempiin ratkaisuihin. Naapuriratkaisuista voidaan valita satunnaisesti jokin arvioitavaksi, mutta jos ratkaisu huononee, määräytyy sen hyväksyminen sekä lämpötilan että ratkaisujen välillä olevan eron mukaan. Haku ei alussa vielä pyri keskittymään hyviin ratkaisuihin vaan hakuavaruu-

den laajempaan kartoittamiseen. Haun tehostaminen tapahtuu lämpötilan vähetessä. [Blum and Roli 2003]

Simuloidulla jäähdytyksellä ja tabuetsinnällä on muutamia olennaisia eroja. Ensinnäkin tabuetsintä vertailee seuraavaa siirtoa tehdessään naapuriratkaisujaan, mutta simuloitu jäädytys ei satunnaisuudessaan tutki naapurustonsa laatua. Toisekseen tabuetsintä huomioi naapuriratkaisujen laadussa muitakin piirteitä kuin vain sen, paljonko ratkaisu paranee kohdefunktion kannalta. Kolmanneksi tabuetsintä ei rajoita siirtymissä ratkaisujen välisiä eroja toisin kuin simuloitu jäähdytys, jossa loppua kohden valituksi tulevissa siirroissa ratkaisu voi huonontua vain vähän. Simuloidussa jäähdytyksessä voidaan hyödyntää tabuetsinnän ajatuksia manipuloimalla strategisesti lämpötilaparametria sen sijaan, että sitä jatkuvasti pelkästään pienennettäisiin. Myös naapuriratkaisujen tabuetsintämäinen arviointi on osoittautunut tehokkaaksi menetelmäksi simuloidussa jäähdytyksessä. Sen sijaan tabuetsinnän muistirakenteita hyödyntämättömät menetelmät eivät ole osoittautuneet kovinkaan kilpailukykyisiksi tavalliseen tabuetsintään verrattuna. Muitakin kuin tässä mainittuja yhdistelmiä kyseisistä metaheuristiikoista on toki mahdollista käyttää. [Glover and Laguna 1997]

4.4.2. Geneettiset algoritmit

Geneettiset algoritmit (genetic algorithms) ovat populaatioperustaisia metaheuristiikkoja ja kuuluvat *evoluutioalgoritmeihin* (evolutional algorithms). Ne perustuvat evoluutiossa tapahtuvaan luonnonvalintaan, jossa parhaiten ympäristöön sopeutuvat yksilöt jäävät henkiin. Kahden tai useamman yksilön risteyttämisellä ja ajoittaisilla mutaatioilla luodaan nykyisen sukupolven populaatioon kuuluvista yksilöistä seuraavan sukupolven populaation yksilöt ja pyritään näin parantamaan ratkaisua. Sopivimmat yksilöt siirretään seuraavaan populaatioon tai niitä käytetään seuraavan sukupolven yksilöiden vanhempina. Yksilöiden ei välttämättä tarvitse olla varsinaisesti ratkaisuja, mutta niiden tulisi jollain tavalla olla muutettavissa ratkaisuiksi. Usein ratkaisut esitetään bittijonoina tai kokonaislukujen joukkoina. [Blum and Roli 2003]

Tabuetsinnän ja geneettisten algoritmien hybridisoinnissa luontevana lähtökohtana voisi olla hyödyntää muistia tallentamalla historiatietoja ratkaisujen ominaisuuksista. Geneettisissä algoritmeissa ei käsitellä siirtoja samalla tavalla kuin tabuetsinnässä, jossa voidaan erotella, mitä sellaista edeltävässä ratkaisussa oli, mikä puuttuu sitä seuraavasta ratkaisusta, ja mitä tässä seuraavassa ratkaisussa on uutta edelliseen verrattuna. Lisäksi tabuetsinnässä joudutaan kiinnittämään enemmän huomiota ratkaistavan ongelman luomaan kontekstiin. Geneettisissä algoritmeissa tällaista ongelman rakenteen tutkimista voidaan

tehdä hyödyntämällä sopivissa kohdissa uusia ratkaisuja tuottavia lokaalin haun menetelmiä kuten tabuetsintää. Geneettisten algoritmien ja tabuetsinnän yhdistämisessä voi olla hyödyllistä koota jäsentyneitä yhdistelmiä ratkaisuista, joilla on tiettyjä haluttuja ominaisuuksia kuten niiden kelvollisuus. [Glover and Laguna 1997]

4.4.3. Sirontahaku

Myös *sirontahaku* (scatter search) ja sen yleistetty muoto *polkujen uudelleenlinkittäminen* (path relinking) ovat populaatioperustaisia metaheuristiikkoja. Sirontahaku alkaa luomalla joukko viitteellisiä ratkaisuja, jotka vastaavat kelvollisia ratkaisuja käsiteltävissä olevaan ongelmaan. Näitä ratkaisuja yhdistelemällä luodaan uusia ratkaisuja, jotka kenties vaativat vielä joitakin korjaavia muutoksia ollakseen kelvollisia. Näihin ratkaisuihin kohdistetaan joitakin niitä parantelevia menetelmiä kuten lokaalia hakua. Seuraavalle iteraatiolle viitteellisten ratkaisujen joukkoon valitaan parhaat näistä uusista parannelluista ratkaisuista sekä edeltävän iteraation viitteellisistä ratkaisuista. Polkujen uudelleenlinkittäminen sen sijaan tutkii ratkaisujen välisiä polkuja hakuavaruudessa ratkaisujen yhdistämiseksi. [Blum and Roli 2003]

Tabuetsinnällä ja sirontahaulla on yhteinen alkuperä, jälkimmäinen nimittäin alun perin käsitettiin lähinnä osana tabuetsinnän puitteita. Sirontahaussa muistin käyttämisellä onkin yhteyksiä muistin hyödyntämiseen tabuetsinnässä. Tabuetsintää voidaan hyödyntää sirontahaussa muun muassa viitteellisten ratkaisujen muodostamisessa ja ratkaisujen parantelemisessa. Polkuja uudelleenlinkittämällä tabuetsinnässä voidaan esimerkiksi etsiä suoraviivaisempia polkuja ratkaisujen välillä verrattuna siihen, millaista reittiä haku on kulkenut. [Glover *et al.* 2000]

4.4.4. GRASP

GRASP (Greedy Randomized Adaptive Search Procedure) on metaheuristiikka, joka hyödyntää iteratiivisesti kahta erilaista vaihetta hakuprosessissa. Haun ensimmäisessä vaiheessa rakennetaan jokin kelvollinen ratkaisu lisäämällä siihen palasia yksi kerrallaan. Mahdolliset lisättävät ratkaisun osat kirjataan kandidaattilistalle, jonka parhaista ehdokkaista valitaan satunnaisesti yksi lisättäväksi ratkaisuun. Kun ratkaisusta on saatu täydellinen, siirrytään seuraavaan vaiheeseen. Koska koottu ratkaisu ei välttämättä ole lokaali optimi, pyritään sitä nyt parantelemaan lokaalin haun avulla. [Feo and Resende 1995] Tässä saatetaan hyödyntää esimerkiksi tabuetsintää tai simuloitua jäähdytystä. Koska perinteinen GRASP ei juurikaan hyödynnä muistia, muut metaheuristiikat osoittautuvat usein sitä tehokkaammiksi. Toisaalta se kaikessa yksinkertaisuu-

dessaan usein kykenee löytämään melko hyviäkin ratkaisuja nopeasti ja on helposti yhdisteltävissä muihin hakustrategioihin. [Blum and Roli 2003]

5. Tabuetsinnän sovelluksia

Tabuetsintää on sovellettu useilla eri aloilla moninaisten optimointiongelmien ratkaisemiseen. Perinteisiin tabuetsinnän sovellusalueisiin kuuluvat graafiteorian ongelmat sekä aikataulutus [Gendreau and Potvin 2010]. Nykyään tabuetsintää hyödynnetään monissa muunkin tyyppisissä optimointiongelmissa. Tarkastellaan seuraavaksi paria perinteistä esimerkkiongelmaa.

5.1. Kauppamatkustajan ongelma

Optimoinnin kenties perinteisin esimerkki *kauppamatkustajan ongelma* (travelling salesman problem) kuuluu graafiteorian ongelmiin. Siinä yritetään löytää kauppamatkustajalle lyhin reitti tiettyjen kaupunkien kautta kulkemiseen niin, että kussakin kaupungissa vieraillaan tasan kerran ja lopuksi palataan takaisin lähtöpisteeseen. Graafina kuvattuna kaupungit ovat graafin solmuja ja kaarien pituudet kertovat näiden kaupunkien välisen etäisyyden. Symmetrisessä kauppamatkustajan ongelmassa graafi voidaan esittää suuntaamattomana, jolloin kaarien pituus on kumpaankin suuntaan sama. Ongelman epäsymmetrisessä versiossa sen sijaan solmujen välillä saattaa esiintyä kaaria vain toiseen suuntaan tai pituudet eri suuntiin voivat erota toisistaan. Kaupunkien määrän lisääntyessä mahdollisten ratkaisujen määrä kasvaa nopeasti hyvin suureksi.

5.2. Ajoneuvojen reititysongelma

Ajoneuvojen reititysongelma (vehicle routing problem) kumpuaa jakelutoiminnan alalta ja on kauppamatkustajan ongelman yleinen muoto. Se on hyvin tunnettu ja paljon tutkittu ongelma kombinatorisessa optimoinnissa. Tabuetsintä on todettu useissa yhteyksissä tehokkaaksi keinoksi pyrkiä ratkaisemaan kyseinen optimointiongelma. [Gendreau and Potvin 2010]

Ajoneuvojen reititysongelma voidaan kuvata suunnattuna tai suuntamattomana graafina, jossa yksi solmuista on varikko, loput kaupunkeja tai asiakkaita ja kaaret näiden välisiä matkoja. Kussakin solmussa käydään vain kerran ja vain yhdellä ajoneuvolla. Lisäksi voi olla muitakin rajoituksia, kuten tietty määrä tavaraa, mikä tiettyyn solmuun täytyy kuljettaa, sekä aika, joka lastin purkamiseen kussakin kohteessa kuluu. Yhteensä matkoihin ja lastin purkamiseen saa kulua tietty määrä aikaa. [Gendreau *et al.* 1994; Taillard *et al.* 1997] Ongelmasta on olemassa useita eri versioita erilaisilla rajoituksilla. Esimerkiksi aikaikkunallisessa reititysongelmassa voidaan ottaa huomioon aikaikkuna, jonka sisällä toimitus tulisi tehdä kohdesolmussa [Taillard *et al.* 1997].

Gendreau ja muut [1994] tutkivat tabuetsinnän tehokkuutta ajoneuvojen reititysongelmassa, jossa rajoitettiin ajoneuvojen kapasiteettia sekä reitin pituutta. Heidän menetelmässään jokin solmu poistetaan reitiltään ja lisätään jollenkin toiselle reitille. Siirrot on mahdollista tehdä niin, että ratkaisut eivät noudata kapasiteetin tai reitin pituuden rajoituksia, mutta tällöin sovelletaan sopivia rangaistuksia. Vertailussa muita menetelmiä vastaan tabuetsintä ja simuloitu jäähdytys osoittautuivat selkeästi tehokkaammiksi kuin perinteiset heuristiikat, jotka kyllä saattoivat olla nopeampia, mutta eivät yltäneet yhtä hyviin ratkaisuihin. Tutkijoiden menetelmä oli tässä vertailussa yleisesti tehokkain.

6. Yhteenveto

Tässä tutkielmassa esiteltiin tabuetsinnäksi kutsuttua optimoinnissa käytettävää metaheuristiikkaa. Liikkeelle lähdettiin yleisistä metaheuristiikan piirteistä, jonka jälkeen keskityttiin tabuetsinnän perusajatuksiin. Tämän jälkeen edettiin esittelemään joitain edistyneempiä tekniikoita ja lopulta annettiin vielä pari esimerkkiä kombinatorisista ongelmista, joita tabuetsinnän ja muidenkin metaheuristiikkojen avulla voidaan pyrkiä tehokkaasti ratkaisemaan.

Tabuetsintä käsittää paljon sellaistakin, mitä tämän tutkielman puitteissa ei ole ollut mahdollista tarkastella. Glover ja Laguna [1997] ovat tehneet varsin laajan katsauksen tabuetsinnästä ja sen piirteistä, jotka vielä näin lähes parikymmentä vuotta myöhemmin ovat aihepiirille relevantteja. Tutkielma keskittyi pääasiassa joihinkin keskeisimpiin ja perinteisimpiin tabuetsinnän lähtökohtiin eikä erityisemmin ota kantaa siihen, miten haku varsinaisesti tulisi toteuttaa. Esittelemättä jääneitä edistyneempiä tekniikoita ovat muun muassa yksinkertaiseen ja joustavaan tabuetsintäkoodiin pyrkivät *yhtenäinen tabuetsintä* (unified tabu search) ja *rakeinen tabuetsintä* (granular tabu search) [Gendreau and Potvin 2010]. Myös erilaisia tabuetsinnän sovellusalueita on valtavasti, ja menetelmän hyödyntäminen on levinnyt yhä laajemmille alueille, kuten resurssijohtamiseen, taloudelliseen ja sijoitussuunnitteluun, terveydenhuoltoon, energia- ja ympäristölinjauksiin, hahmontunnistukseen ja bioteknologiaan. [Glover and Laguna 2013]

Tabuetsintä on osoittautunut useissa yhteyksissä tehokkaaksi menetelmäksi. Valitettavasti aina tabuetsinnän hyödyntämisessä ei onnistuta. Tämä johtuu usein siitä, että menetelmän soveltajat eivät kunnolla ymmärrä tabuetsinnän perimmäisiä piirteitä tai vaihtoehtoisesti käsittelemäänsä ongelmaa. [Gendreau and Potvin 2010] Tutkielma ei myöskään ota kantaa siihen, mikä optimointimenetelmä tai metaheuristiikka on missään mielessä paras. Muutamia muitakin menetelmiä esiteltiin lyhyesti tutkielmassa. Tabuetsintä itsessään on osoittautunut tehokkaaksi erinäisten ongelmien ratkaisemisessa, mutta myös sen

hybridisointi muiden metaheuristiikkojen kanssa on mielenkiintoinen tutkimusalue. Tabuetsinnän tutkimus vaikuttaa olevan edelleen kiivasta ja menetelmästä lieneekin vielä paljon opittavaa.

Viiteluettelo

Roberto Battiti and Giampietro Tecchiolli. 1994. The reactive tabu search. ORSA Journal on Computing 6, 2, 126-140.

Christian Blum, Jakob Puchinger, Günther R. Raidl, and Andrea Roli. 2011. Hybrid metaheuristics in combinatorial optimization: A survey. Applied Soft Computing 11, 6, 4135-4151.

Christian Blum and Andrea Roli. 2003. Metaheuristics in combinatorial optimization: Overview and conceptual comparison. ACM Computing Surveys 35, 3, 268-308.

Thomas A. Feo and Mauricio G. C. Resende. 1995. Greedy randomized adaptive search procedures. Journal of Global Optimization 6, 2, 109-133.

Michel Gendreau, Alain Hertz, and Gilbert Laporte. 1994. A tabu search heuristic for the vehicle routing problem. Management Science 40, 10, 1276-1290.

Michel Gendreau and Jean-Yves Potvin. 2005. Metaheuristics in combinatorial optimization. Annals of Operations Research 140, 1, 189-213.

Michel Gendreau and Jean-Yves Potvin. 2010. Tabu search. In: Michel Gendreau and Jean-Yves Potvin (eds.), Handbook of Metaheuristics. Springer, 41-60.

Fred Glover. 1989. Tabu search – part 1. ORSA Journal on Computing 1, 3, 190-206.

Fred Glover. 1990. Tabu search: a tutorial. Interfaces 20, 4, 74-94.

Fred Glover and Manuel Laguna. 1997. Tabu Search. Kluwer Academic Publishers.

Fred Glover and Manuel Laguna. 2013. Tabu search*. In: Panos M. Pardalos, Ding-Zhu Du, and Ronald L. Graham (eds.), Handbook of Combinatorial Optimization. Springer, 3261-3362.

Fred Glover, Manuel Laguna, and Rafael Martí. 2000. Fundamentals of scatter search and path relinking. Control and Cybernetics 29, 3, 653-684.

Éric Taillard, Philippe Badeau, Michel Gendreau, François Guertin, and Jean-Yves Potvin. 1997. A tabu search heuristic for the vehicle routing problem with soft time windows. Transportation Science 31, 2, 170-186.

El-Ghazali Talbi. 2002. A taxonomy of hybrid metaheuristics. Journal of Heuristics 8, 5, 541-564.

El-Ghazali Talbi. 2009. Metaheuristics: from Design to Implementation. John Wiley & Sons.

Eloquent ORM Laravelissa

Marko Pellinen

Tiivistelmä.
Tässä tutkielmassa käsitellään Laravel-sovelluskehyksen Eloquent ORMin käyttöä sekä yleisellä tasolla ORM-menetelmää. Tutkimuksen menetelminä olivat kirjallisuuskatsaus sekä empiirinen tutkimus, joka perustui PHP-kehitysympäristö Laravel 5.1:n käyttöön web-sivujen toteutuksessa. Tutkielman aineistona olivat tieteellinen kirjallisuus sekä Laravelin internetissä sijaitseva dokumentaatio. ORM-menetelmän käyttöä tarkastellaan käytännön esimerkkien avulla Eloquent ORMilla toteutetulla esimerkkitietokannalla. Toisena tarkasteltavana relaatiotietokantojen käsittelytapana on Laravelissa myöskin oletuksena oleva fluent query builder -rajapinta. ORM-menetelmän hyötyjä ovat helppokäyttöisyys, ohjelmakoodin yksinkertaisuus, Eloquent ORMin malliluokkien suhteiden hyödyntäminen sekä automaattinen SQL-injektiolta suojautuminen. Näistä automaattinen SQL-injektiolta suojautuminen sisältyy myös fluent query builderiin. Suurin ORM-menetelmän haitta on N+1-ongelma, josta johtuvat myös kyselyjen suoritusajan kasvu ja tietokannan kuormittuminen. ORM-menetelmällä ei ole myöskään mahdollista toteuttaa kaikkein monimutkaisimpia SQL-kyselyitä. Tätä puutetta tosin voidaan paikata olio-ohjelmoinnin sovelluslogiikassa. Tutkimuksen tuloksien mukaan ORM-menetelmä soveltuu hyvin suurimpaan osaan relaatiotietokantojen käsittelytehtävistä. Joissakin tapauksissa kannattaa kuitenkin käyttää muita menetelmiä varsinkin, jos prosessointiajalla ja kyselyjen tehokkuudella on suuri merkitys.

Avainsanat ja -sanonnat: object-relational mapping, olio, relaatiotietokanta, PHP, Laravel, Eloquent ORM.

1. Johdanto

Pysyvyys, tiedon säilyminen ohjelman suorituksesta toiseen, on tärkeässä roolissa nykyaikaisissa sovelluksissa. Useimmiten pysyvyyden toteuttamiseen käytetään relaatiotietokantoja. Ohjelmien toimintalogiikka toteutetaan kuitenkin usein olio-ohjelmointina. [Russell 2008]

Relaatiotietokannat pohjautuvat matemaattisiin teoreettisiin malleihin, kun taas olioparadigma pohjautuu enimmäkseen tietojenkäsittelyn periaatteisiin [Pekkala 2004; Pop and Altar 2014]. Näiden kahden eri periaatteisiin pohjautuvan mallin yhdistäminen saattaa olla hankalaa. Tätä ongelmaa helpottamaan on kehitetty ORM-menetelmä [Russell 2008].

Lyhenne ORM tulee sanoista Object-Relational Mapping. Tällä tarkoitetaan ohjelmalogiikan olioiden kuvaamista tietokannan relaatioiksi. Tällöin tietokannan tauluja ja rivejä voidaan käsitellä ohjelmassa olioina.

Tässä tutkielmassa tarkastellaan ORM-menetelmän eri osa-alueita ohjelmistokehys Laravel 5.1:n Eloquent ORMia esimerkkinä käyttäen. Tutkielma sisältää esimerkkitietokannan, jonka avulla tietokannanhallinnan eri vaiheita esitellään. Lisäksi tarkastellaan myös fluent query builder -rajapinnan käyttöä yhdessä Eloquent ORMin kanssa.

Luku 2 käsittelee ORM-menetelmään liittyviä yleisiä asioita. Luvussa 3 esitellään esimerkkitietokanta, jonka avulla Eloquent ORMin käyttöä tutkielmassa havainnollistetaan. Luvussa 4 käsitellään migraatiotiedostoja, joilla varsinainen tietokanta luodaan tietokantajärjestelmään. Luku 5 puolestaan käsittelee model-luokkien määrittämisen perusasioita. Model-luokkien käsittelyä jatketaan luvussa 6, jossa käydään läpi olioiden suhteiden kuvaaminen. Luku 7 käsittelee tietokantaan tallentamista Eloquent ORMin avustuksella. Luvussa 8 käsitellään erilaisia tietokantaan kohdistuvia kyselyitä. Näiden lisäksi luvussa on myös for each -silmukan, laiskan latauksen sekä ahnaan latauksen käsittely. Luvussa 9 esitellään metodit, joita käytetään tietokannasta poistamiseen Eloquent ORMilla. Lopuksi luvussa 10 on pohdinta tutkielman aiheesta.

2. Yleistä

Sovellukset käyttävät tietoa, jonka täytyy pysyä tallessa ohjelmien eri ajokertojen välillä. Tämä toteutetaan usein relaatiotietokannan avulla. Relaatiotietokanta muodostuu tauluista, jotka sisältävät dataa, joka on jaoteltu riveihin ja sarakkeisiin. Tätä dataa voidaan käsitellä CRUD-operaatioilla. CRUD-akronyymi tulee sanoista Create, Read, Update ja Delete [Wikipedia 2016]. CRUD-operaatioilla tietokannan tietoja voidaan siis luoda, lukea, päivittää ja poistaa.

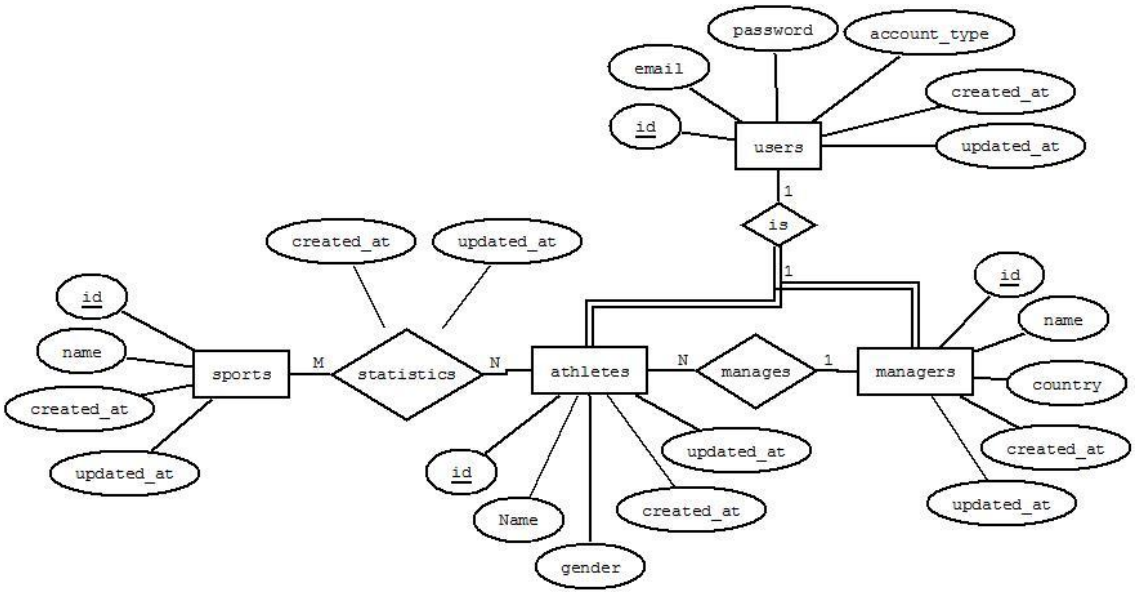
Useimmiten sovelluksien toimintalogiikka luodaan käyttämällä olio-ohjelmointia. Tällöin ohjelman suoritus koostuu luokista luoduista oliosta. Yksi nykyaikaisten sovellusten tärkeimmistä osa-alueista on olioiden ja tietokannan välinen tiedonsiirto. Tähän tehtävään helpoin vaihtoehto on käyttää ORM-menetelmää. ORM-menetelmän avulla sovelluskehittäjät voivat kuvata olemassa olevat tietokannan taulut model-luokkina ja näistä luokista luoduilla olioilla käsitellä tietokannan rivejä ohjelmalogiikassa olioparadigman mukaan. Tällöin olion attribuutit vastaavat taulun sarakkeita. Olioiden suhteet kuvataan taulujen vierasavaimiin tai suhdetauluun. ORM-menetelmän käyttö vähentää huomattavasti tietokantayhteyden muodostamiseen tarvittavan ohjelmistokoodin määrää ja parantaa täten myös sovelluskehittäjien tehokkuutta. [Richardson

2008] ORM-menetelmän avulla myös CRUD-operaatioiden toteuttamisesta tulee yksinkertaisempaa.

ORM-menetelmää käytetään usein sovelluskehysten yhteydessä, kuten Django tai Laravel. Nämä ovat web-sovelluskehyksiä. Ohjelmointikielenä Laravelissa on PHP. Eloquent ORM sisältyy Laravel-sovelluskehykseen. Tässä tutkielmassa havainnollistetaan Eloquent ORMin käyttöä.

3. Esimerkkitietokanta

Esimerkkitietokantana tutkielmassa on tietokanta urheilijoista, joilla voi olla manageri. Urheilijat ja managerit ovat sovelluksen käyttäjiä. Urheilijoilla voi olla myös lajeja, joihin he osallistuvat. Tietokannan ER-kaavio on kuvassa 1.



Kuva 1. Esimerkkitietokannan ER-kaavio.

Kuvan 1 ER-kaavion mukaan tietokanta on jaettu tauluiksi seuraavasti: *users*, *managers*, *athletes*, *sports* sekä *statistics*. Näistä kaikilla muilla, paitsi *statistics*-taululla, on pääavaimena **id**-attribuutti. *Statistics*-taulu on *athletes*- ja *sports*-taulun välinen suhdetaulu, joten sen pääavaimena on vierasavaimista koostuva komposiittiavain (**athlete_id, sport_id**). Kaikilla tauluilla on aikaleima attribuutit **created_at** ja **updated_at**.

Vierasavaimien suhteen on noudatettu seuraavaa nimeämiskäytäntöä: aluksi viitatun taulun nimi yksikkömuodossa, jonka jälkeen alaviiva ja lopuksi id. *Managers*-taululla on vierasavaimena **user_id**. *Athletes*-taululla on vierasavaimina **user_id** sekä **manager_id**. Attribuutilla **manager_id** voi olla tyhjäarvoja. *Statistics*-taululla on vierasavaimina **athlete_id** sekä **sport_id**. Lisäksi *users*-taululla on yksikäsitteisyysavaimena (unique key) **email**-attribuutti ja *sports*-taululla **name**-attribuutti.

Muita attribuutteja tauluilla ovat *users*-taululla **password** ja **account_type**, *managers*-taululla **name** ja **country** sekä *athletes*-taululla **name** ja **gender**.

4. Migraatiotiedostot

Laravelissa tietokanta luodaan migraatiotiedostojen avulla. Näissä migraatiotiedostoissa määritellään tietokannan taulujen kuvaukset. Migraatiotiedostoja ei tule sekoittaa model-luokkiin, joissa taas kuvataan luokkana jo olemassa olevat tietokannan taulut. Migraatiotiedostojen rakennetta selventää liite 1, jossa on kuvattuna esimerkkitietokannan migraatiot. Usein migraatiot jaetaan eri tiedostoihin, mutta esimerkkitietokannan tapauksessa kaikki migraatiot ovat samassa tiedostossa. Silloin kun migraatiot sijaitsevat eri tiedostoissa, voidaan näiden tiedostojen avulla muuttaa tietokannan rakennetta tilasta toiseen. Migraatiotiedostoja voidaankin ajatella eräänlaisena versionhallintajärjestelmänä tietokannalle [Laravel 2015].

Tietokantaan luodaan määrittelyjen mukaiset taulut ajamalla migraatiotiedostot komennolla **php artisan migrate**. Tämä luo lisäksi tietokantaan **migrations**-taulun, joka pitää kirjaa ajetuista migraatiotiedostoista. Migraatioissa voidaan mennä taaksepäin palauttamalla tietokanta aiempaan rakenteeseen komennolla **php artisan migrate:rollback**. Migraatiotiedostot sisältävät *up()*- ja *down()*-funktion. *Up()*-funktio ajetaan suoritettaessa migraatio (**migrate**), kun taas *down()*-funktio ajetaan peruutettaessa migraatio (**rollback**). [Laravel 2015]

Laravelin kanssa yhteensopivat relaatiotietokantajärjestelmät ovat tällä hetkellä MySQL, Postgres, SQLite ja SQL Server [Laravel 2015]. Samat migraatiotiedostot eivät välttämättä toimi eri järjestelmissä samalla tavalla, vaan tiedostoihin saattaa joutua tekemään pieniä muutoksia. Tämä johtuu tietokantajärjestelmissä olevista pienistä eroista. Tämä kannattaa muistaa siirryttäessä tietokantajärjestelmästä toiseen tai suunnitellessa alkuperäisiä migraatioita.

Migraatiotiedostot ovat hyvin virheherkkiä, ja pienikin virhe migraatiossa voi johtaa tilanteeseen, jossa tilasta toiseen siirtyminen ei enää onnistu. Tällöin ratkaisuna on usein muokata tietokannan tauluja manuaalisesti. Migraatioiden hyöty on kuitenkin kiistaton silloin, kun sovelluksella on monta eri kehittäjää. Tällöin kehittäjät pystyvät pitämään tietokannan rakenteen samanlaisina paikallisissa kehitysympäristöissään. Migraatioiden toiminta täytyy testata erittäin huolellisesti ennen niiden viemistä tuotantoympäristöön, tai seurauksena saattaa olla sovelluksen seisonta-aika (down time) [Vermolen et al. 2011].

5. Model-luokkien määrittäminen

Eloquent ORMia käytettäessä sovellukseen luodaan model-luokkia, jotka mallintavat tietokannan tauluja. Näiden model-luokkien yliluokka on Laravelissa

oletuksena oleva **Model**-luokka, jonka ominaisuudet model-luokat perivät. Liitteessä 2 on esimerkkitietokannan model-luokkien ohjelmakoodi kokonaisuudessaan. Luokka nimetään yleensä yksikkömuodossa, esimerkiksi **User**. Tietokannan taulut taas nimetään tavallisesti monikkomuodossa, esimerkiksi **users**. Tällöin model-luokassa ei tarvitse erikseen kertoa, mihin tauluun luokka viittaa. Luokassa voidaan kuitenkin käyttää *protected*-tyyppistä muuttujaa nimeltä **$table** nimeämiskäytännön ohittamiseksi. Tällöin muuttujalle annetaan arvoksi tietokannan taulun nimi. [Laravel 2015] Jos esimerkiksi **Phone**-luokka viittaisikin **smartphones**-tauluun, sijoitettaisiin **Phone**-luokkaan lause:

```
protected $table = 'smartphones'; .
```

Käyttäjän määritettävissä olevat relaation attribuutit sijoitetaan *protected*-tyyppiseen taulukkomuuttujaan nimeltä **$fillable** [Laravel 2015]. Esimerkkitietokannan **User**-luokan **$fillable**-taulukon alkioille arvot annetaan seuraavasti:

```
protected $fillable = ['email', 'password', 'ac
count_type']; .
```

Model-luokissa käytetään joskus myös toista *protected*-tyyppistä muuttujaa nimeltä **$hidden**. Tällä muuttujalla määritetään ne relaation attribuutit, jotka halutaan pitää piilossa mallista luoduilta taulukoilta tai JSON-lomakkeilta. [Laravel 2015] Esimerkkitietokannan **User**-luokassa se määritellään näin:

```
protected $hidden = ['password']; .
```

Model-luokkiin voidaan luoda myös rajoitteita (scope), jotka ovat yleisiä kyseisen taulun kohdalla. Tällainen rajoite luodaan funktiona, joka saa parametrina kyselyolion (**$query**). Funktio myös palauttaa kyselyolion, johon on ketjutettu kyselyä tarkentavia metodeita. [Laravel 2015] Rajoitteita luodaan esimerkin 1 tavalla.

```
public function scopeFemales($query)
{
   return $query->where('gender', '=', 1);
}
```

Esimerkki 1. Rajoitteen luonti.

Model-luokissa kuvataan myös relaatioiden ja olioiden suhteet, näistä tarkemmin luvussa 6.

6. Suhteiden kuvaaminen model-luokissa

Tietokannan relaatioiden (ts. taulujen) välillä on suhteita. Myös olio-ohjelmoinnin olioiden välillä voi olla suhteita. Tämän luonnollinen seuraus on, että käsiteltäessä relaatiota olioina relaatioiden suhteet voidaan mallintaa myös

olioiden suhteina. [Russell 2008] Eri suhdetyypit ovat: 1:1, 1:n ja m:n [Pekkala 2004].

Relaatioiden väliset suhteet toteutetaan vierasavainten avulla. Suhteessa 1:1 vierasavain sijoitetaan toiseen suhteen tauluista. Periaatteessa ei ole väliä, kumpaan tauluun se sijoitetaan, kunhan tarvittava liitos voidaan tehdä. Suhde 1:n toteutetaan taas siten, että suhteen n-puolen tauluun lisätään vierasavain viittaamaan suhteen 1-puolen tauluun. Suhdetta m:n varten taas tarvitaan ylimääräinen suhdetaulu. [Pekkala 2004]

Suhdetyypissä 1:1 relaation yhden rivin vierasavainsarakkeen arvo vastaa toisen relaation yhden rivin pääavainsarakkeen arvoa, lisäksi vierasavainsarakkeella on yksikäsitteisyysrajoite. Suhdetyypissä 1:n n-puolen vierasavaimen arvo vastaa yhtä 1-puolen pääavaimen arvoa. Suhdetyypin m:n suhdetaulussa on vierasavaimet, joiden arvot vastaavat suhteen taulujen pääavaimia [Pekkala 2004]. Suhdetaulussa vierasavaimet muodostavat lisäksi kaksiarvoisen pääavaimen.

Laravelin Eloquent ORMissa suhteet toteutetaan lisäämällä tietokannan taulua mallintavalle model-luokalle funktio, joka toteuttaa luokkien eli olioiden välisen suhteen [Laravel 2015]. Tämän havainnollistamiseksi seuraavaksi on esimerkkejä siitä, kuinka eri suhdetyypit toteutetaan esimerkkitietokannan model-luokissa. Esimerkeissä 2 ja 3 yksi käyttäjä voi olla urheilija ja urheilija on yksi käyttäjä. Esimerkeissä 4 ja 5 managerilla voi olla monta urheilijaa, mutta yhdellä urheilijalla voi olla maksimissaan yksi manageri. Esimerkeissä 6 ja 7 urheilijalla voi olla monta lajia ja samaa lajia voi harrastaa monta urheilijaa.

```
public function athlete()
{
   return $this->hasOne('App\Athlete');
}
```

Esimerkki 2. Yhden suhde yhteen pääavaimen sisältävän relaation puolella (**User**-luokka).

```
public function user()
{
   return $this->belongsTo('App\User');
}
```

Esimerkki 3. Yhden suhde yhteen viiteavaimen sisältävän relaation puolella (**Athlete**-luokka).

```
  public function athlete()
  {
     return $this->hasMany('App\Athlete');
  }
```

Esimerkki 4. Yhden suhde moneen pääavaimen sisältävän relaation puolella (**Manager**-luokka).

```
  public function manager()
  {
     return $this->belongsTo('App\Manager');
  }
```

Esimerkki 5. Yhden suhde moneen viiteavaimen sisältävän relaation puolella (**Athlete**-luokka).

```
  public function sports()
  {
     return $this->belongsToMany('App\Sport', 'statis
tics')->withTimestamps();
  }
```

Esimerkki 6. Monen suhde moneen urheilijan mallissa (**Athlete**-luokka).

```
  public function athlete()
  {
     return $this->belongsToMany('App\Athlete',
'statistics')->withTimestamps();
  }
```

Esimerkki 7. Monen suhde moneen lajin mallissa (**Sport**-luokka).

Suhdetaulu nimetään tavallisesti taulut yksikkömuodossa aakkosjärjestyksessä alaviivalla erotettuna. Tässä tapauksessa taulun nimi olisi **athlete_sport**. Tällöin Eloquent ORM osaisi automaattisesti käyttää oikeaa taulua urheilijoiden ja lajien välisenä suhdetauluna. Esimerkeissä 6 ja 7 *belongsToMany()*-metodille annetaan kuitenkin toisena parametrina **statistics**-taulu, jolloin tätä taulua käytetään suhdetauluna. Eloquent ORM ei oletuksena käytä aikaleimoja suhdetaululle. Esimerkkitietokannassa on kuitenkin aikaleimat myös suhdetau-

lulle. Tämän johdosta suhdetta luotaessa joudutaan kutsumaan *withTimestamps()*-metodia. [Laravel 2015]

Eloquent ORMissa on mahdollista luoda suhde myös kahden model-luokan välille, jotka eivät ole suoraan liitoksissa toisiinsa. Esimerkkitietokannassa tämä on toteutettu luomalla suhde model-luokkien **Sport** ja **User** välille. Näiden luokkien välissä on **Athlete**-luokka, joka täytyy antaa toisena parametrina *hasManyThrough*()-metodille. Ensimmäisenä parametrina *hasManyThrough*()-metodille annetaan kohdeluokka, eli tässä tapauksessa **User**-luokka. Tällöin kutsumalla **Sport**-luokan toteuttavan olion kautta *users*()-funktiota, saadaan taulukkona kyseisen lajin kaikkien urheilijoiden käyttäjien oliot. [Laravel 2015] Suhde määritellään esimerkissä 8 esitellyllä tavalla.

```
  public function users() {
     return $this->hasManyThrough('App\User',
'App\Athlete');
  }
```

Esimerkki 8. Has Many Through -suhteen määritys.

7. Tallennus tietokantaan Eloquent ORMilla

PHP-ohjelmointikieli sisältää kaksi operaattoria olioiden ja luokkien ominaisuuksien käsittelyyn. Operaattoria nuoli (->) käytetään silloin, kun halutaan kutsua olion attribuuttia tai olion metodia. Toinen operaattori taas on tuplakaksoispiste (::), jota käytetään kutsuttaessa staattista attribuuttia tai staattista metodia. [Stack Overflow 2016] Operaattoria tuplakaksoispiste (::) kutsutaan myös heprean kielisellä nimellä Paamayim Nekudotayim [PHP-Manual 2016]. Model-luokista luotuja olioita käsitellään näiden operaattorien avulla.

7.1. Olioiden tallennus tietokantaan

Model-luokista luodut oliot edustavat tietokannan rivejä. Olion attribuutit taas edustavat tietokannan sarakkeita. [Richardson 2008] Asettamalla olioiden attribuuteille arvot voidaan nämä tallentaa myöhemmin tietokantaan *save()*-metodin avulla [Laravel 2015]. Esimerkissä 9 luodaan $**user**-olio ja annetaan attribuuteille arvot. Tämän jälkeen olio tallennetaan tietokantaan *save()*-metodilla.

```
$user = new User;
$user->email = 'jose@gmail.com';
$user->password = 'diipadaapa';
$user->account_type = 'manager';
$user->save();
```

Esimerkki 9. Olion luominen ja tallentaminen tietokantaan.

Esimerkistä 9 voidaan huomata, että vain kolme **$user**-olion attribuuteista tarvitsee määritellä. Loput attribuuteista Eloquent ORM osaa määritellä automaattisesti. Nämä muut attribuutithan ovat yksilöivä **id** sekä aikaleimat **created_at** ja **updated_at**. Samaan lopputulokseen päästäisiin käyttämällä myös **User**-luokan *create()*-funktiota esimerkin 10 tavalla [Laravel 2015].

```
$user = User::create([
    'email' => 'jose@gmail.com',
    'password' => 'diipadaapa,
    'account_type' => 'manager',
]);
```

Esimerkki 10. Tietokantaan rivin lisääminen *create()*-funktiolla.

Esimerkissä 10 käytetään **Model**-yliluokasta perittyä *create()*-funktiota, joka tallentaa parametrina annetun taulukon sisältämät tiedot suoraan tietokantaan sekä palauttaa näistä luodun model-olion **$user** [Laravel 2016]. Tämän olion käsittelyä voitaisiin jatkaa toimintalogiikassa, ja myöhemmin tallentaa olio uudelleen tietokantaan olion *save()*-funktiolla. Tällöin *save()*-funktio päivittäisi tietokannassa jo olemassa olevan rivin tietoja [Laravel 2015].

7.2. Suhteiden tallennus tietokantaan

Eloquent ORMissa tietokannan relaatioiden suhteet on määritelty funktioina model-luokissa. Näiden funktioiden avulla voidaan tietokantaan myös tallentaa suhteet toteuttavien vierasavaimien ja suhdetaulujen tiedot. Tämä tapahtuu seuraavalla tavalla: kutsutaan olion suhdefunktiota, jonka jälkeen kutsutaan tämän suhdetyypin toteuttavan luokan funktiota ja annetaan tälle funktiolle parametriksi suhteen toinen olio tai rivien pääavain **id**:t [Laravel 2016]. [Laravel 2015] Esimerkissä 11 tallennetaan tietokantaan urheilijalle manageri. Näiden välinen suhdetyyppihän on yhden suhde moneen. Esimerkissä 11 on käytetty myös *find()*-metodia, jonka avulla luodaan model-luokan toteuttava olio pääavain **id**:n perusteella.

```
$athlete = Athlete::find(3);
$manager = Manager::find(2);
$manager->athletes()->save($athlete);
```

Esimerkki 11. *Save()*-metodin käyttö.

Esimerkissä 11 käytetään funktiota *save()* suhteen tallentamiseen tietokantaan. Tällöin managerin **id**-attribuutti tallentuu **athletes**-taulun **manager_id**-sarakkeeseen. Tässä tapauksessa *save()*-funktiota täytyy kutsua **$manager**-olion puolelta. Tämä johtuu siitä, että suhteen toteuttava **HasMany**-luokka sisältää *save()*-metodin. **$athlete**-olion puolella suhteen toteuttava **BelongsTo**-luokka ei taas sisällä *save()*-metodia. *Save()*-metodi sisältyy myös luokkiin **HasOne** ja **BelongsToMany**. [Laravel 2016] Luvussa 6 määriteltiin näiden avulla model-luokkien suhteet.

Suhdetyypeissä 1:1 ja 1:n voidaan vierasavaimia käsitellä myös *associate()*- ja *dissociate()*-metodeilla, jotka sisältyvät **BelongsTo**-luokan valikoimaan [Laravel 2016]. Urheilijan ja managerin tapauksessa näitä metodeita kutsuttaisiin tällöin **$athlete**-olion puolelta. Pelkkä *associate()*-metodin käyttö ei kuitenkaan vielä tallenna muutoksia tietokantaan, vaan sen jälkeen täytyy **$athlete**-oliolle kutsua vielä *save()*-metodia. *Dissociate()*-metodin avulla vierasavain voidaan poistaa **athletes**-taulusta. Myös *dissociate()*-metodin kutsumisen jälkeen täytyy **$athlete**-oliolle kutsua vielä *save()*-metodia muutoksien viemiseksi tietokantaan. [Laravel 2015] Esimerkissä 12 havainnollistetaan näiden kahden metodin käyttöä.

```
$athlete->manager()->associate($manager);
$athlete->save();

$athlete->manager()->dissociate();
$athlete->save();
```

Esimerkki 12. *Associate()*- ja *dissociate()*-metodien käyttö.

Suhdetyypin m:n tapauksessa on *save()*-funktion lisäksi myös muita funktioita rivien käsittelyyn suhdetaulussa. Näitä ovat esimerkiksi metodit *attach()* ja *detach()*. *Attach()*-metodi tallentaa rivin suhdetauluun, kun taas *detach()*-metodilla voidaan poistaa rivi suhdetaulusta. Lisättäessä vain yhtä riviä suhdetauluun voidaan *attach()*-metodille antaa parametrina olio. Jos käsitellään suhdetaulun useampaa riviä, täytyy *attach()*- ja *detach()*-metodeille antaa parametrina taulukko, jossa on listattuna olioiden pääavain **id**:t. Kutsuttaessa pelkkää *detach()*-funktiota poistaa tämä kaikki olioon liittyvät rivit suhdetaulusta. [Laravel 2015] Esimerkissä 13 luetellaan erilaisia tapoja käyttää *attach()*- ja *detach()*-funktioita.

```
$john = Athlete::find(2);
$brenda = Athlete::find(3);
$sport = Sport::find(2);
$sport->athletes()->attach($john);
$sport->athletes()->detach($john);
$sport->athletes()->attach([2, 3]);
$sport->athletes()->detach([3]);
$sport->athletes()->detach();
```

Esimerkki 13. *Attach()*- ja *detach()*-metodien käyttö.

Attach()- ja *detach()*-metodit sisältyvät luokkaan **BelongsToMany**. Tämä luokka sisältää myös metodin *sync()*, jonka voidaan ajatella yhdistävän sekä *attach()*- että *detach()*-metodin toiminnan. *Sync()*-funktio poistaa suhdetaulusta kaikki muut olioon liittyvät rivit paitsi ne, jotka sille annetaan parametrina. *Sync()*-funktio ottaa parametrina vain taulukon, joka sisältää pääavain **id**:t, eikä ollenkaan olioita. [Laravel 2015] Esimerkissä 14 suhdetauluun lisätään yhdelle lajille kolme urheilijaa ja poistetaan muut mahdolliset urheilijat *sync()*-funktion avulla.

```
$sport->athletes()->sync([2, 4, 5]);
```

Esimerkki 14. *Sync()*-metodin käyttö.

Myös metodilla *saveMany()* voidaan tallentaa monia suhteita samanaikaisesti. Tämä metodi ottaa parametrina vain taulukon, jossa on olioita. [Laravel 2015] Esimerkissä 15 suhdetauluun lisätään kaksi urheilijaa s*aveMany()*-metodin avulla.

```
$sport->athletes()->saveMany([$john, $brenda]);
```

Esimerkki 15. Suhteiden tallentaminen *saveMany()*-metodilla.

8. Kyselyt tietokannasta

Laravelissa on fluent query builder -rajapinta, jonka avulla voidaan toteuttaa useimmat kyselyt tietokannasta. Eloquent ORM sisältää taas omat menetelmänsä kyselyjen toteuttamiseen. Eloquent ORMin hakuihin voidaan yhdistää fluent query builderin metodeita ketjuttamalla metodit nuolioperaattorilla (->). Fluent query builderin syntaksissa käytetään tietokannan taulujen nimiä, kun taas Eloquent ORMin syntaksissa kyselyn rakentaminen aloitetaan model-luokan nimestä. Sekä fluent query builder että Eloquent ORM muodostavat ketjutetun lauseen perusteella SQL-kyselyn PHP:n PDO-lisäosaa käyttäen. PDO-lisäosa käsittelee attribuuttien arvot parametreina, mikä torjuu mahdolliset SQL-injektioyritykset [PHP-Manual 2016]. [Laravel 2015]

8.1. Kyselyt fluent query builder -rajapinnan avulla

Fluent query builder tarjoaa SQL-lauseenosia vastaavia metodeita. Näitä ovat esimerkiksi: *select()*, *where()* ja *join()*. Haku aloitetaan kutsumalla **DB**-luokan *table()*-metodia, jolle annetaan parametrina taulun nimi, josta kysely lähtee liikkeelle. Tämä vastaa SQL-kielen from-osaa. Sen jälkeen kyselyyn voidaan ketjuttaa muita metodeita. Jos halutaan suorittaa haku tietokannasta, kutsutaan joko *first()*- tai *get()*-metodia. Nämä metodit käynnistävät varsinaisen haun. *First()*-metodi palauttaa oliona kyselyn tuloksesta ensimmäisen rivin, kun taas *get()*-metodi palauttaa oliotaulukkona kaikki kyselyn tulosrivit. [Laravel 2015] Esimerkissä 16 muodostetaan kysely, joka hakee esimerkkitietokannasta urheilijan, jonka id on 3, lajien id:n ja nimen.

```
   DB::table('athletes')->join('statistics', 'athletes.id',
'=', 'athlete_id')->join('sports', 'sport_id', '=',
'sports.id')->where('athletes.id', '=', 3)-
>select('sports.id AS id', 'sports.name AS name')->get();
```

Esimerkki 16. Urheilijan lajien haku fluent query builderilla.

Vaikka fluent query builderillä tehdyt kyselyt palauttavatkin olioita, ei näiden olioiden ominaisuuksia voi samalla tavalla muokata ja sen jälkeen päivittää tietokantaan, kuten Eloquent ORMin model-luokista luotujen olioiden tapauksessa. Fluent query builderissa rivin päivitys täytyy tehdä kyselyn yhteydessä ketjuttamalla kyselyyn *update()*-metodi. *Update()*-metodille annetaan parametrina taulukko, joka sisältää avain ja arvo -pareina päivitettävät tiedot. [Laravel 2015] *Update()*-metodia käytetään esimerkin 17 tavalla.

```
   DB::table('managers')->whereName('Pep')-
>update(['country' => 'Germany', 'name' => 'Hans']);
```

Esimerkki 17. *Update()*-metodin käyttö.

8.2. Kyselyt Eloquent ORMin avulla

Eloquent ORMin model-luokkia voidaan myös käyttää tietojen hakuun tietokannasta. Tähän tehtävään sopivia metodeita ovat muun muassa *find()* ja *all()*. Kyselyssä lähdetään liikkeelle luokan nimestä, jonka jälkeen tulee Paamayim Nekudotayim -operaattori ja kutsuttava metodi. Tämän jälkeen lisättävät metodit ketjutetaan tutusti nuolioperaattorilla. Metodien *find()* ja *all()* jälkeen ei enää tarvitse kutsua *first()*- tai *get()*-metodia paitsi, jos kyselyn ketjuttamista jatketaan fluent query builderin -metodeilla. [Laravel 2015] Tuloksena saatavien olioiden tyyppi pysyy model-luokkien mukaisena silloinkin, kun kyselyyn ketjutetaan fluent query builderin metodeita. Näiden olioiden attribuuttien arvoja voidaan sitten muutella toimintalogiikassa, kuten esimerkissä 9 tehtiin.

Model-luokista muodostettujen olioiden suhteita voidaan kätevästi käyttää myös kyselyissä. Jos suhdetta kutsutaan metodina, täytyy tällöin käyttää myös *first()*- tai *get()*-metodia. Suhdetta voidaan kutsua myös ilman metodisulkuja, jolloin Eloquent ORM palauttaa kokoelman tai kokoelmataulukon. Kokoelma on Laravelin tietotyyppi, joka perustuu PHP:n taulukkoon. Kokoelmaan on kuitenkin lisätty ominaisuuksia, joita taulukossa ei ole. [Laravel 2015] Esimerkissä 18 suoritetaan kysely, jossa haetaan urheilijan, jonka **id** on 3, lajit. Kysely on siis käytännössä sama kuin esimerkin 16 kysely. Eloquent ORMin ansiosta kysely on kuitenkin huomattavasti helpompi toteuttaa.

```
Athlete::find(3)->sports;
```

Esimerkki 18. Urheilijan lajien hakeminen Eloquent ORMilla.

Luvun 5 esimerkissä 1 esiteltiin rajoitteiden määritys funktioina model-luokissa. Myös näitä funktioita voidaan ketjuttaa hakulauseen jatkoksi. Esimerkin 1 rajoitefunktiota voidaan kutsua ilman **scope**-etuliitettä esimerkin 19 tavalla. Tämä haku hakisi kaikki naisurheilijat, joiden managerin **manager_id** on 2.

```
Athlete::whereManagerId(2)->females()->get();
```

Esimerkki 19. Model-luokan rajoitefunktion käyttö.

8.3. Where-metodit

Where-metodeille on tässä oma kohta, koska niiden käyttämisessä on eniten variaatiota verrattaessa esimerkiksi *select()*- ja *join()*-metodeihin, joiden käyttäminen taas on melko suoraviivaista ja tulee hyvin ilmi esimerkistä 16. Fluent query builder tarjoaa monenlaisia where-metodeita hakujen toteuttamiseksi. Eri where-metodit jäljittelevät vastaavia SQL-lauseen where-osan ehtoja. Fluent query builderissa tavallinen where-metodi määritellään näin:

```
where('name', '=', 'Brenda') .
```

Tällöin ensimmäisenä parametrina on attribuutin nimi, jonka perusteella kysellään. Toisena parametrina on vertailuoperaattori, jota kyselyssä käytetään. Kolmantena parametrina on arvo, johon attribuutin arvoa verrataan. [Laravel 2015]

Fluent query builderissa voidaan where-metodeita rakentaa myös lisäämällä where-hakusanaan attribuutin nimi isolla alkukirjaimella, esimerkiksi näin: *whereId()* tai *whereName()*. Tällaiselle where-metodille annetaan parametrina pelkästään attribuutin arvo. Jos taas attribuutin nimi on esimerkiksi **user_id**, voidaan where-metodi kirjoittaa muotoon *whereUserId()*. Lisäksi yhdellä where-metodilla voidaan hakea monen attribuutin perusteella lisäämällä attribuuttien väliin and-sana isolla alkukirjaimella. Tämä tapahtuu seuraavasti:

```
whereNameAndGender('Brenda', 1); .
```

Muita yleisesti käytettäviä where-metodeita ovat *orWhere()*, *whereNull()*, *whereNotNull()*, *whereBetween()*, *whereNotBetween()*, *whereIn()* ja *whereNotIn()*. Metodille *orWhere()* annetaan samat parametrit kuin tavallisellekin *where()*-metodille. Metodiin *orWhere()* ei kuitenkaan voi lisätä attribuuttien nimiä metodin jatkoksi samalla tavalla kuin where-hakusanan tapauksessa. Metodit *whereNull()* ja *whereNotNull()* ottavat parametrina vain attribuutin nimen. Metodeille *whereBetween()* ja *whereNotBetween()* annetaan kaksi parametria: attribuutin nimi sekä taulukko, jossa on ala- ja yläraja arvojen välille. Myös metodeille *whereIn()* ja *whereNotIn()* annetaan kaksi parametria: attribuutin nimi sekä taulukkona kaikki arvot, jotka kyselyyn sisällytetään. [Laravel 2015]

8.4. For each -silmukka

Verrattaessa esimerkkejä 16 ja 18 keskenään voidaan havaita, kuinka paljon helpompaa kyselyn tekeminen voi olla käytettäessä model-luokkien suhteita. Käytännössä suhteiden hyödyntäminen ei kuitenkaan aina ole aivan näin helppoa ja nopeaa. Tämä johtuu siitä, että suhdetta voidaan kutsua vain yksittäiselle oliolle. Kyselyn palauttaessa taulukon joudutaan se käymään läpi for each -silmukalla [Laravel 2015]. Esimerkissä 19 haetaan kaikki käyttäjät, jotka ovat urheilijoita. Tämän jälkeen tallennetaan urheilijoiden nimet for each -silmukassa taulukkoon **$names,** taulukon avaimena käyttäjän **id** ja arvona urheilijan nimi (**name**-attribuutti).

```
$users = User::whereAccountType('athlete')->get();
$names = array();

foreach($users as $user) {
  $names[$user->id] = $user->athlete->name;
}
```

Esimerkki 19. Käyttäjien nimien tallennus taulukkoon.

8.5. Laiska lataus ja ahnas lataus

Esimerkissä 19 haetaan urheilijat ja näiden nimet laiskalla latauksella (lazy loading). Tällöin ensin suoritetaan kysely tietokannasta, jossa haetaan kaikki käyttäjät, jotka ovat urheilijoita. Tämän jälkeen for each -silmukassa suoritetaan jokaiselle näistä käyttäjistä kysely tietokannasta, jossa haetaan athlete-suhteen kautta urheilijan nimi. Tällöin suoritetaan siis niin monta kyselyä tietokannasta, kuin mitä urheilijoita on, ja lisäksi vielä yksi kysely. Jos urheilijoita olisi 1000 kappaletta, suoritettaisiin 1001 hakua. Näin monen kyselyn suorittaminen kuormittaa tietokantaa. Tätä sanotaan N+1-ongelmaksi ja se johtuu laiskasta lataamisesta. [Laravel 2015]

Eloquent ORMissa on mahdollista käyttää myös ahnasta latausta (eager loading) hakujen suorittamiseen tietokannasta. Ahnas lataus toteutetaan *with()*-metodin avulla. Tälle metodille annetaan parametriksi suhde, joka ahnaalla latauksella haetaan. Toteutettaessa esimerkkiä 19 vastaava haku ahnaalla latauksella suoritetaan vain kaksi kyselyä riippumatta siitä, kuinka monta urheilijaa tietokannassa on. Ensimmäisessä kyselyssä haetaan kaikki **users**-taulun rivit, joissa **account_type**-attribuutin arvo on 'athlete'. Toisessa kyselyssä taas haetaan kaikki **athletes**-taulun rivit. [Laravel 2015] Esimerkissä 20 käytetään ahnasta latausta esimerkkiä 19 vastaavan haun toteuttamiseen. Tällöin for each -silmukassa ei enää suoriteta montaa hakua tietokannasta, vaan tiedot on ladattu ennalta muistiin.

```
   $users = User::whereAccountType('athlete')-
>with('athlete')->get();
   $names = array();

   foreach($users as $user) {
     $names[$user->id] = $user->athlete->name;
   }
```

Esimerkki 20. Athlete-suhteen haku ahnaalla latauksella.

9. Tietokannasta poistaminen

Eloquent ORMilla voidaan tietenkin myös poistaa tietokannasta rivejä. Tähän tehtävään sopivat metodit *delete()* ja *destroy()*. *Delete()*-metodia käytetään siten, että suoritetaan haku käyttämällä model-luokkaa, jonka jälkeen kutsutaan *delete()*-metodia. *Destroy()*-metodia käytettäessä ei hakua tietokannasta suoriteta, vaan rivit poistetaan suoraan tietokannasta. Parametriksi *Destroy()*-metodi ottaa pääavain **id**:tä, jotka voidaan antaa joko yksittäin tai taulukkona. [Laravel 2015] Esimerkissä 21 esitellään näiden kahden metodin käyttöä.

```
   Athlete::destroy(3);
   Athlete::destroy([2, 4, 5]);
   Athlete::where('id', 'like', '%%')->delete();
   User::where('account_type', '=', 'athlete')->delete();
```

Esimerkki 21. *Delete()*- ja *destroy()*-metodien käyttö.

10. Pohdinta

Tässä tutkielmassa käsiteltiin tärkeimpiä Eloquent ORMin perusasioita sekä yleisellä tasolla ORM-menetelmää. Tutkielmassa esiteltiin esimerkkitietokanta, joka luotiin migraatioilla. Tämän jälkeen käsiteltiin vastaavien model-luokkien

määrittämistä. Lopuksi esimerkkitietokantaa käsiteltiin sekä fluent query builderin että Eloquent ORMin avulla.

Sovelluksia tehdessä tietokannan käsittely on usein haastavaa ja työlästä. Eloquent ORM on suunniteltu helpottamaan tätä prosessia monella tavalla. Eloquent ORM muodostaa tietokantayhteyden tarvittaessa, jolloin sovelluskehittäjien ei tarvitse tästä erikseen huolehtia, vaan he voivat keskittyä sisällön tuottamiseen. ORMia käytettäessä joidenkin kehittäjien ei tarvitse välttämättä edes ymmärtää tietokannan käsittelystä mitään, vaan he voivat keskittyä muun ohjelmakoodin tekemiseen.

ORM-menetelmän suurin hyöty on relaatiotietokannan tietojen käsittelyn mahdollistaminen nykyaikana yleisesti käytössä olevan olio-ohjelmointiparadigman konventioiden mukaan. Muita ORM-menetelmän hyötyjä ovat helppokäyttöisyys, ohjelmakoodin yksinkertaisuus, Eloquent ORMin malliluokkien suhteiden hyödyntäminen sekä automaattinen SQL-injektiolta suojautuminen. ORM-menetelmää käytettäessä tulee ottaa huomioon laiskan latauksen mahdollisesti aiheuttama tietokannan kuormittuminen, jota voidaan ehkäistä käyttämällä ahnasta latausta. Näin vältetään N+1-ongelma.

Olio-ohjelmista yhteyden saamiseksi relaatiotietokantaan on olemassa myös muita tapoja ORM-menetelmän lisäksi, mutta ne eivät kuitenkaan hyödynnä ohjelmointikielen olio-ominaisuuksia [Russell 2008]. Russell [2008] toteaa: ”jotta olio-ohjelmoinnista saataisiin täysi hyöty irti, täytyy tietokantayhteyden muodostavan teknologian noudattaa seuraavia periaatteita: vastuualueiden erottelu, tiedon kätkentä, periytyminen, muutoksien havaitseminen, yksilöiminen sekä riippumattomuus tietokannasta.”

ORM-menetelmä sopii hyvin suurimpaan osaan relaatiotietokantojen käsittelytehtävistä.

Viiteluettelo

Laravel 5.1 API. 2016. https://laravel.com/api/5.1/. Checked 18.1.2016.

Laravel 5.1 documentation. 2015. https://laravel.com/docs/5.1. Checked 8.11.2015.

Lauri Pekkala. 2004. *Pysyvyyden toteuttaminen Java-sovelluksissa erityisesti ORM-menetelmän avulla*. Pro gradu -tutkielma. Tietojenkäsittelytieteiden laitos, Tampereen yliopisto.

PHP-Manual. 2016. http://php.net/manual/en/. Checked 15.1.2016.

Dragos-Paul Pop and Adam Altar. 2014. Designing an MVC model for rapid web application development. *Procedia Eng.* 69, 1172-1179.

Chris Richardson. 2008. ORM in dynamic languages. *ACM Queue* 6, 3, 28-37.

Graig Russell. 2008. Bridging the object-relational divide. *ACM Queue* 6, 3, 16-26.

Stack Overflow. 2016. Where do we use the object operator "->" in PHP?. http://stackoverflow.com/questions/3037526/where-do-we-use-the-object-operator-in-php. Checked 20.1.2016.

Sander D. Vermolen, Guido Wachsmuth and Eelco Visser. 2011. Generating database migrations for evolving web applications. *Proceedings of the 10th ACM International Conference on Generative Programming and Component Engineering*. 83-92.

Wikipedia. 2016. Create, read, update and delete. https://en.wikipedia.org/wiki/Create,_read,_update_and_delete. Checked 29.1.2016.

Esimerkkitietokannan migraatiotiedosto

```
<?php

use Illuminate\Database\Schema\Blueprint;
use Illuminate\Database\Migrations\Migration;

class CreateTables extends Migration
{
    // Run the migrations. @return void
    public function up()
    {
      // Contains user's data
      Schema::create('users', function (Blueprint $table) {
        $table->increments('id');
        $table->string('email')->unique();
        $table->string('password', 60);
        $table->enum('account_type', array('athlete', 'manager'));
        $table->timestamps();
      });

      // Contains manager's data
      Schema::create('managers', function (Blueprint $table) {
        $table->increments('id');
        $table->string('name');
        $table->string('country');
        $table->timestamps();
        $table->integer('user_id')->unsigned()->unique();
        $table->foreign('user_id')->references('id')->on('users');
      });

      // Contains athlete's data
      Schema::create('athletes', function (Blueprint $table) {
        $table->increments('id');
        $table->string('name');
        $table->integer('gender')->unsigned();
        $table->timestamps();
        $table->integer('manager_id')->unsigned()->nullable();
        $table->foreign('manager_id')->references('id')->on('managers');
        $table->integer('user_id')->unsigned()->unique();
        $table->foreign('user_id')->references('id')->on('users');
      });

      // Contains sport's data
      Schema::create('sports', function (Blueprint $table) {
        $table->increments('id');
        $table->string('name')->unique();
        $table->timestamps();
      });
```

```
    // Contains athletes's sport specific data
    Schema::create('statistics', function (Blueprint $table) {
      $table->integer('athlete_id')->unsigned()->index();
      $table->integer('sport_id')->unsigned()->index();
      $table->primary(['athlete_id', 'sport_id']);
      $table->foreign('athlete_id')->references('id')->on('athletes');
      $table->foreign('sport_id')->references('id')->on('sports');
      $table->timestamps();
    });
  }

  // Reverse the migrations. @return void
  public function down()
  {
    Schema::drop('statistics');
    Schema::drop('sports');
    Schema::drop('athletes');
    Schema::drop('managers');
    Schema::drop('users');
  }
}
```

Esimerkkitietokannan model-luokat

User.php

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    //The database table used by the model.
    protected $table = 'users';

    //The attributes that are mass assignable.
    protected $fillable = ['email', 'password', 'account_type'];

    //The attributes excluded from the model's JSON form.
    protected $hidden = ['password'];

    // One-to-one relationship
    public function athlete()
    {
      return $this->hasOne('App\Athlete');
    }

    // One-to-one relationship
    public function manager()
    {
      return $this->hasOne('App\Manager');
    }
}
```

Manager.php

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Manager extends Model
{
    // Table managers
    protected $table = 'managers';

    //The attributes that are mass assignable.
    protected $fillable = ['user_id', 'name' , 'country'];
```

```
    // One-to-one relationship
    public function user()
    {
      return $this->belongsTo('App\User');
    }

    // One-to-many relationship
    public function athletes()
    {
      return $this->hasMany('App\Athlete');
    }
}
```

Athlete.php

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Athlete extends Model
{
    // Table athletes
    protected $table = 'athletes';

    // Mass-assignables
    protected $fillable = ['user_id', 'name' , 'gender', 'manager_id'];

    // One-to-one relationship
    public function user()
    {
      return $this->belongsTo('App\User');
    }

    // One-to-many relationship
    public function manager()
    {
      return $this->belongsTo('App\Manager');
    }

    // Many-to-many relationship
    public function sports()
    {
      return $this->belongsToMany('App\Sport', 'statistics')->withTimestamps();
    }

    // Scope for usual query
    public function scopeFemales($query)
```

```
    {
      return $query->where('gender', '=', 1);
    }

    // Scope for usual query
    public function scopeMales($query)
    {
      return $query->where('gender', '=', 0);
    }
}
```

Sport.php

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Sport extends Model
{
    // Table sports
    protected $table = 'sports';

    // Mass-assignables
    protected $fillable = ['name'];

    // Many-to-many relationship
    public function athletes()
    {
      return $this->belongsToMany('App\Athlete', 'statistics')-
>withTimestamps();
    }

    // Has-many-through relationship
    public function users() {
      return $this->hasManyThrough('App\User', 'App\Athlete');
    }
}
```

Statistic.php

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Statistic extends Model
```

```
{
    // Table statistics
    protected $table = 'statistics';

    // Table's primaryKey for model
    protected $primaryKey = ['athlete_id', 'sport_id'];

    // Mass-assignables
    protected $fillable = ['athlete_id', 'sport_id'];
}
```

Konvoluutioverkot kuvantunnistuksessa

Jenni Saaristo

Tiivistelmä.
Konvoluutioverkot ovat koneoppimisen menetelmä, joka sopii erityisen hyvin kuvantunnistukseen ja muihin sellaisiin tehtäviin, jotka ovat rakenteeltaan voimakkaasti hierarkkisia ja joiden ratkaisussa on mielekästä irrottaa ja hyödyntää paikallisia piirteitä. Tässä tutkielmassa esitellään konvoluutioverkon osat, arkkitehtuuri sekä toiminta kuvantunnistustehtävässä.

Avainsanat ja -sanonnat: Neuroverkot, konvoluutioverkot, kuvantunnistus.

1. Johdanto

Kohteiden tunnistaminen kuvasta on vaikea tietojenkäsittelytieteellinen ongelma. Se ei ole täysin ilmeistä, sillä jokapäiväisen kokemuksemme mukaan mikään ei ole niin helppoa kuin katsominen. Tämä helppous on kuitenkin äärimmäisen tehokkaan ja pääosin automaattisen näköjärjestelmämme luoma illuusio. Todellisuudessa arkisten ja luonnollisten kuvien tunnistaminen on tehtävä, joka on kaukana yksinkertaisten luokittelijoiden ulottumattomissa [LeCun *et al*. 2015]. Voi esimerkiksi pohtia, millaisella pikseliarvoja vertailevalla säännöllä kaikki kuvan 1a esittämät kuviot tunnistettaisiin kahvikupeiksi mutta toisaalta erotettaisiin esimerkiksi keittokulhoista ja säilyketölkeistä (kuvat 1b ja 1c).

Kuva 1. ImageNet-aineiston kuvia luokista a) kahvikuppi (mug), b) keittokulho (soup bowl) ja c) säilyketölkki (canned food).

Konenäön ongelmia on ratkottu nyt jo puoli vuosisataa, mutta yleisiä, luotettavasti toimivia ratkaisuja sen keskeisimpiin tehtäviin ei vieläkään ole löydetty [Krüger *et al.* 2013]. Tekoälyn historiaa luonnehtii pieneltä vaikuttaneiden pulmien kasvaminen ylittämättömiksi esteiksi ongelmakentän laajentuessa leikkiongelmista todelliseen dataan [Russell and Norvig 2010, 21], ja sama pätee myös konenäköön. Tietyissä hyvin rajoittuneissa käyttötarkoituksissa keinotekoiset näköjärjestelmät toki jo toimivat hyvin ja jopa yli-inhimillisellä tarkkuudella, mutta biologisen näön joustavuudesta ja yleisyydestä ollaan vielä kaukana [Krüger *et al.* 2013].

Tällä hetkellä yksi parhaista ratkaisuista *kuvantunnistustukseen* (object recognition), joka on konenäön eräs aliongelma, ovat syväoppivat konvoluutiooperaatiota hyödyntävät neuroverkot eli *konvoluutioverkot* (convolutional networks) [LeCun *et al.* 1989]. Neuroverkkojen kokoa on perinteisesti rajannut neuronien lisäämisestä seuraava opittavien parametrien räjähdysmäinen kasvu sekä opettamisen vaikeutuminen, mutta konvoluutioverkkojen kohdalla ongelma on niiden rakenteen vuoksi huomattavasti pienempi. Konvoluutioverkot ovatkin olleet ensimmäisiä käyttökelpoisiksi osoittautuneita syviä neuroverkkoarkkitehtuureita [LeCun *et al.* 2015]: niissä on tyypillisesti yli seitsemän kerrosta, ja myöhemmin puheena olevassa Szegedyn ja kumppaneiden [2014] GoogLeNet-verkossa on kerroksia jopa 22.

Tässä tutkielmassa esitellään konvoluutioverkon perusrakenne ja sen käyttö kuvantunnistusongelmassa. Aluksi tutustutaan hieman itse ongelmaan sekä ImageNet-kuvatietokantaa hyödyntävään kilpailuun, jonka puitteissa viimeaikaisia menestyksiä on todistettu [Russakovsky *et al.* 2015]. Sen jälkeen käydään läpi konvoluutioverkon rakenne, ja lopuksi nähdään, miten eräs hyvin menestynyt verkko hahmottaa syötekuviaan.

Syväoppiminen on aiheena liian laaja tässä työssä esiteltäväksi. Syvyyteen liittyvät tekijät käsitellään sellaisina kuin ne esittäytyvät nimenomaisesti konvoluutioverkkojen yhteydessä ottamatta kantaa siihen, missä määrin ne soveltuvat muunlaisiin syväoppiviin verkkoihin.

Lukijan oletetaan omaavan perustiedot neuroverkoista. Tarvittaessa suositellaan tutustumaan alan kursseihin ja perusteoksiin [mm. Karpathy 2015; Juhola 2015; Haykin 2009].

2. Kuvantunnistus

2.1. Kuvantunnistuksen haasteet

Rajoittamaton visuaalinen haku ja tunnistus on NP-täydellinen ongelma [Tsotsos 1990]. Vaikeaa siitä tekee syöteavaruuden valtava koko, kun jokainen ku-

van pikseli tulkitaan omaksi parametrikseen ja usein vielä värikanavien määrällä kerrottuna. Vain 32x32 pikselin kokoisen kolmikanavaisen värikuvan määrittely vaatii 3072 parametria ja käytännön sovelluksissa yleisempi 320x320 värikuva jo 30 720 parametria. Jälkimmäisessä syöteavaruudessa on siis 30 720 ulottuvuutta, joiden arvot voivat vaihdella vapaasti toisistaan riippumatta. Huomautettakoon, että jo 400 ulottuvuudessa, vaikka jokainen ulottuvuus voisi saada arvokseen vain joko 0 tai 1, kaikkien mahdollisten kuvioiden tallennus sekä siinä joukossa suoritettava haku ovat tehtäviä, joiden vaativuus ylittää selvästi kaikki käytännöllisyyden rajat [Davies 2005, 4]. On selvää, ettei kuvantunnistusta ratkaista raa'an voiman menetelmillä.

Ulottuvuuksien vähentäminen on siksi kuvantunnistusjärjestelmän keskeisimpiä tehtäviä. Käyttötarkoitus ohjaa vähennystä: tärkeää on erotella tunnistamisen kannalta olennaiset elementit epäolennaisesta vaihtelusta, kuten valaistuksesta tai asennosta [LeCun *et al*. 2015]. Olennaisiksi tulkittuja elementtejä kutsutaan *piirteiksi* (features), ja ne muodostavat järjestelmän sisäisen mallin eli *representaation* [Bengio *et al.* 2015, Ch. 1]. Yleisesti ottaen tekoälysovelluksen onnistuminen riippuu pitkälti sen käyttämän representaation sopivuudesta sille asetetun ongelman ratkaisemiseen [Haykin 2009, 24; Bengio *et al.* 2015, Ch. 1]. Hyvien piirteiden synnyttäminen on siis ensiarvoisen tärkeää.

Tekoäly- ja kuviontunnistusjärjestelmien kehityksessä on pitkään keskitytty pienten, tarkkarajaisten ongelmien ratkaisuun käsin valikoitujen piirteiden pohjalta [LeCun *et al*. 2015; Bengio *et al*. 2015, Ch. 1]. Tämäntyyppiset menetelmät ovatkin usein myös konenäön kirjallisuuden keskiössä (esimerkiksi Davies [2005]): kuvantunnistuksen alalla paljon käytettyjä ovat esimerkiksi piirrepankit SIFT ja HOG [Fisher *et al.* 2014, 260; 123]. Tällaiseen luovaan valikointiin perustuvat järjestelmät saattavat olla hyvinkin toimivia, erityisesti olosuhteiltaan hallituissa teollisuusympäristöissä, mutta niiden rakentaminen on vaikeaa, hidasta ja erityisosaamista vaativaa käsityötä [LeCun *et al.* 2015] ja sovellusala usein rajallinen. Varsinainen rajoittamaton kuvantunnistus vaatisi todennäköisesti yleisemmän menetelmän, joka oppisi omat piirteensä ja siten myös representaationsa. Tällaista vain vähäistä esityötä vaativaa ja koneoppimista painottavaa lähestymistapaa kutsutaan *representaation oppimiseksi* (representation learning) [Bengio *et al*. 2015, Ch. 1].

Luokittelevan kuvantunnistuksen tekee lisäksi vaikeaksi se, että eri luokkia edustavat kohteet saattavat tietyissä tilanteissa näyttää hyvinkin samanlaisilta, ja toisaalta luokkien sisällä voi olla merkittävää vaihtelua, kuten kuva 1 havainnollisti. Koska jatkossa keskitytään lähinnä kuvien luokitteluun, määritellään seuraavaksi, mitä sillä tarkoitetaan.

2.2. Kuvantunnistuksen lajeja

Kuvantunnistuksessa on yleisesti ottaen kyse kuvassa olevien kohteiden tunnistamisesta [Fisher *et al*. 2014, 192–193]. Tarkemmin ottaen voidaan pyrkiä kohteen paikantamiseen, luokitteluun tai molempiin, ja joskus jopa kolmiulotteisen muodon rekonstruoimiseen [Russell and Norvig 2010, 957].

Kuvan tulkinta niin keinotekoisessa kuin luonnollisessakin visuaalisessa järjestelmässä alkaa useimmiten kuvan *segmentoinnilla* eli mahdollisten kohteiden paikantamisella ja eristämisellä ympäristöstään [Davies 2005, 103; Farah 2000, 50–51]. Konenäön alalla on kehitetty monenlaisia segmentoinnin menetelmiä, mutta niitä ei tämän tutkielman puitteissa esitellä. Jatkossa oletetaan, ellei muuta sanota, että tunnistettavat kuvat on tarvittaessa jollakin asianmukaisella menetelmällä segmentoitu potentiaalisiin kohteisiin.

Itse tunnistusvaiheessa voitaisiin siis hakea jotakin tiettyä kohdetyyppiä, esimerkiksi jalankulkijoita, tai useita kohdetyyppejä, jotka myös luokitellaan, kuten vaikkapa liikennemerkkejä. Molempia tehtäviä nimitetään usein *kohteen havaitsemiseksi* (object detection). Mikäli olennaista on nimenomaan paikantaa kohde kuva-alasta, puhutaan myös *kohteen paikannuksesta* (object localization) [Fisher *et al*. 2014, 192]. Näiden ero ei kuitenkaan ole kovin selvä. Lisäksi joskus määritellään erikseen tilanne, jossa alkuperäisen kuvan katsotaan esittävän vain yhtä kohdetta tai kategoriaa, jolloin kysymys segmentoinnista voidaan pitkälti ohittaa. Tästä käytetään useimmiten nimitystä *kuvan luokittelu* (image classification) [Karpathy 2015; Russakovsky *et al*. 2015], ja sama käytäntö omaksutaan myös tässä tutkielmassa. Termiä kuvantunnistus (object recognition) käytetään näiden kaikkien lajien yläkäsitteenä.

Robotiikassa keskeisen kohteen tarkan kolmiulotteisen muodon ja asennon päättelemisen ei tässä kirjoituksessa katsota kuuluvan kuvantunnistuksen alle. Valinta perustuu siihen, että kädellisten näköjärjestelmässä tietoista tunnistusta ja toisaalta toiminnan ohjausta palvelevat visuaaliset järjestelmät ovat toisistaan erilliset [Goodale 2011] ja luultavasti käyttävät ylimmillä tasoillaan toisistaan poikkeavia representaatioita [Krüger *et al*. 2013]. Sen perusteella ei siis ole syytä olettaa, että yksittäinen, vain yhtä ylimmän tason representaatiota käyttävä järjestelmä pystyy tehokkaasti tuottamaan sekä luokituksen että tarkan kolmiulotteisen kuvauksen, vaikka alemmilla tasoilla nämä tehtävät jakaisivatkin monia yhteisiä piirteitä.

2.3. Kuvanluokittelun vertailuaineistoja

Yksinkertaisten ja keinotekoisten ongelmien parissa tehty tutkimus ei aina anna kovin hyvää kuvaa siitä, miten tekoälyjärjestelmät selviytyvät todellisista ongelmista [Russell and Norvig 2010, 21]. Siksi hyvät ja laajat, arkisesta maailmas-

ta peräisin olevat aineistot ovat tutkimuksen kannalta tärkeitä. Erityisen merkittäviä ovat avoimet vertailuaineistot, joilla järjestelmien keskinäistä paremmuutta voidaan mitata ja näin ohjata tutkimusta hedelmällisiin suuntiin [Russakovsky *et al.* 2015].

Paljon käytettyjä kuvanluokittelun vertailuaineistoja ovat olleet mm. MNIST, joka käsittää 10 luokkaa ja 70 000 yksittäistä kuvaa käsinkirjoitetuista numeroista, Caltech 101, joka käsittää 101 luokkaa ja 9 144 kuvaa yleisistä kohteista, ja Caltech 256, joka on kuten Caltech 101, mutta sisältää 256 luokkaa ja 30 608 kuvaa [LeCun *et al*. 1998; Russakovsky *et al*. 2015]. Kohteen havaitsemisen ja lokalisoinnin saralla eräs suhteellisen laaja kilpailuaineisto on PASCAL VOC , joka käsittää 20 luokkaa ja kymmeniä tuhansia kuvia. Sen pohjalta järjestetyt vuosittaiset haasteet kuitenkin lopetettiin vuonna 2012 [Everingham *et al*. 2015].

ImageNet on PASCAL VOC:in seuraaja, ja tällä hetkellä suurin täysin nimetty kuvantunnistusaineisto 21 841:llä luokallaan – kuvia aineistossa on kaikkiaan hieman yli 14 miljoonaa [ImageNet]. Vuodesta 2010 lähtien sen pohjalta on vuosittain järjestetty ImageNet Large Scale Visual Recognition Challenge (ILSVRC) -kuvantunnistuskilpailu, jossa opetusaineistona kuvanluokitustehtävässä toimii 1000:n luokan ja yli 1,2 miljoonan yksittäisen kuvan otos [Russakovsky *et al.* 2015].

ILSVRC pitää sisällään useita eri kategorioita, joista pisimpään mukana ovat olleet kuvan luokittelu (vuodesta 2010), yhden kohde-esiintymän paikannus (vuodesta 2011) ja kaikkien kohde-esiintymien paikannus (vuodesta 2013) [Russakovsky *et al.* 2015]. Koska tässä työssä ei ole tarkoitus puuttua segmentoinnin ongelmiin ja menetelmiin, keskitytään jatkossa enimmäkseen yksinkertaiseen kuvanluokitukseen.

Kuvanluokitustehtävässä jokaista aineiston kuvaa vastaa yksi oikea luokitus [Russakovsky *et al.* 2015]. Kuvia ei kuitenkaan ole erityisesti rajattu kohteisiinsa, vaan ne voivat olla hyvinkin laajoja yleiskuvia tai toisaalta pelkkiä yksityiskohtia. Tästä aiheutuu tietenkin toisinaan epäselvyyksiä kuvan varsinaisesta kohteesta, joten arvioinnissa huomioidaankin parhaan arvauksen lisäksi myös viisi todennäköisintä luokkaa, ja vuodesta 2012 alkaen arvioinnissa on yksinkertaisuuden vuoksi nojauduttu vain jälkimmäisiin tuloksiin [Russakovsky *et al.* 2015]. Kisan historian aikana kuvanluokitustehtävän virheprosentti on pudonnut 28,2%:sta 6,7%:iin [Russakovsky *et al.* 2015].

ILSVRC:n aineisto on täysin *luokiteltu* (labeled), joten se mahdollistaa *ohjattujen oppimismenetelmien* (supervised learning) käytön. ILSVRC:lla on ollut huomattava merkitys konvoluutioverkkojen menestyksessä.

3. Konvoluutioverkot

Tässä luvussa esitellään konvoluutioverkko ja selitetään, miten se eroaa perinteisistä neuroverkoista. Samalla tulee tutuksi tutkielmassa omaksuttu nimeämiskäytäntö. Konvoluutioverkko kuvaillaan sekä yhden kerroksen että kokonaisarkkitehtuurin tasolla, sillä jälkimmäisellä on sen toiminnan kannalta huomattavan suuri merkitys. Lopuksi esitetään joitakin huomioita syvien konvoluutioverkkojen opettamisesta.

3.1. Neuroverkoista konvoluutioverkkoihin

Perinteinen eteenpäinsyöttävä neuroverkko on hajautettu tiedonkäsittelijä, joka koostuu useasta kerroksesta yksinkertaisia laskentayksiköitä eli *neuroneita* (units, neurons) sekä kerrosten välisistä yhteyksistä, joiden kautta neuronit saavat syötteensä [Haykin 2009, 2]. *Täysin yhdistetyssä* (fully connected, FC) verkossa jokainen neuroni saa syötteen jokaiselta edellisen kerroksen neuronilta [Haykin 2009, 23].

Yhteydet toimivat hieman samaan tapaan kuin biologisten hermosolujen synapsit, ja jokaiseen yhteyteen liittyy aina *painokerroin* (synaptic weight), jota verkko oppimisen myötä säätää [Haykin 2009, 2]. Painokertoimia kutsutaan usein myös *parametreiksi*. Syötteen saadessaan neuroni laskee siitä painotetun summan ja lisää siihen *kynnysarvon* (bias), minkä jälkeen lopullinen tuloste lasketaan neuronille määritellyn *aktivaatiofunktion* avulla [Haykin 2009, 10–12]. Joitakin yleisesti käytettyjä aktivaatiofunktioita ovat sigmoidi ja hyperbolinen tangentti [Haykin 2009, 14] sekä uudempana tulokkaana *tasasuunnattu lineaariyksikkö* (rectified linear unit) eli ReLU:

$$f(x) = \max(0, x).$$

Näistä ReLU on osoittautunut käytännössä tehokkaimmaksi [Krizhevsky *et al.* 2012] ja tullut varsin suosituksi, vaikka silläkin on tietyt puutteensa [Bengio *et al.* 2015, Ch. 6.3.1].

Tekoälyn menetelmänä neuroverkot ovat houkuttelevia, sillä biologisten aivojen olemassaolon perusteella tiedämme, että hajautetut prosessoijat pystyvät huomattaviin älyllisiin suorituksiin. Neuroverkot ovat kuitenkin hyvin hitaita opetettavia [Juhola, 66–67] ja vaativat myös huomattavia määriä laadukasta opetusdataa [Juhola, 184–185]. Pääasiassa aineiston tarve syntyy siitä, että liian pieni opetusaineisto suhteessa parametrien määrään johtaa verkon *ylioppimiseen* (overfitting) [Haykin 2009, 164–166]. Täysin yhdistetyssä eli FC-neuroverkossa verkon opittavien painoarvojen määrä kasvaa eksponentiaalisesti neuronien määrän funktiona, joten ylioppimisen vaara rajoittaa neuronien määrä voimakkaasti. Suuren parametrimäärän opettaminen, etenkin jos ne ovat useissa kerroksissa, tietysti vaatii myös paljon aikaa – eikä silti aina onnistu. Opittavien

parametrien määrän rajoittaminen on siis yleisesti ottaen hyvä strategia, ja perinteisesti se on toteutettu neuronien ja kerrosten määrää vähentämällä.

Parametrien määrää voidaan kuitenkin vähentää myös suoraan, neuronien määrään koskematta, jos täydellisestä yhdistämisestä luovutaan. Juuri näin konvoluutioverkoissa tehdäänkin. Neuronienkin määrää niissä toki rajoitetaan, mutta yleisesti ottaen konvoluutioverkossa voi olla neuroneita ja kerroksia enemmän kuin mitä FC-verkoissa pidetään käytännöllisenä.

Konvoluutioverkossa parametrien vähennyksiä ei tehdä satunnaisesti, vaan ne sisältävät etukäteistietoa ja oletuksia sekä syöteavaruudesta että ongelmasta [Haykin 2009, 203; LeCun *et al*. 1989]. Tämä etukäteistieto ei kuitenkaan ole luonteeltaan aivan samanlaista kuin piirteiden käsinvalikoinnissa käytetty asiantuntijatieto: konvoluutioverkon pohjana olevat oletukset ovat huomattavasti yleisempiä, joten myös sen sovellusalueet ovat laajemmat.

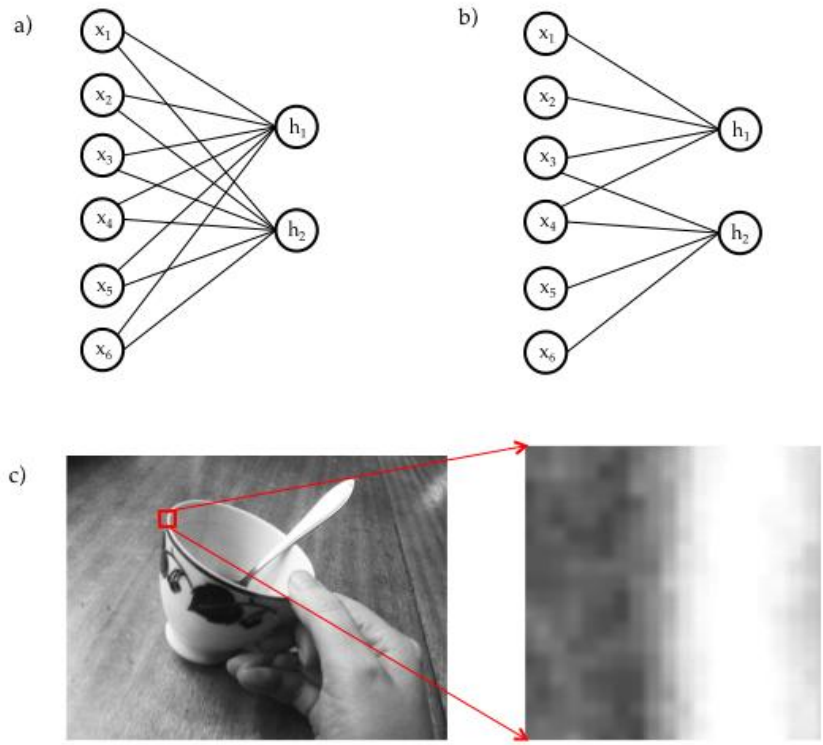
Seuraavaksi tarkastellaan, miten parametrien ja neuronien määriä rajoitetaan konvoluutioverkossa sekä yksittäisen kerroksen että kokonaisarkkitehtuurin tasolla. Esimerkkikerrokseksi otetaan verkon ensimmäinen, tässä tapauksessa kuvasyötettä katseleva konvoluutiokerros, sillä sen visualisoiminen on helpointa. Periaatteessa kaikki konvoluutiokerrokset kuitenkin toimivat samaan tapaan – niiden syöte vain on hieman erilainen.

3.2. Opittavien parametrien vähentäminen

Konvoluutioarkkitehtuurissa opittavien parametrien eli painoarvojen määrää rajoitetaan jokaisen konvoluutiokerroksen tasolla. Sen saavat aikaan kaksi erillistä rakenteellista ratkaisua [Haykin 2009, 29]:

1. Neuronit eivät ole yhdistettyjä kaikkiin edellisen kerroksen neuroneihin (kuten FC-kerroksessa), vaan ne ovat *harvaan yhdistettyjä* (sparsely connected, SC).
2. Painokertoimet eivät vaihtele täysin vapaasti, vaan ne ovat osittain toisiinsa sidottuja – tätä kutsutaan *painokertoimien jakamiseksi* (weight sharing).

Harva yhdistäminen tarkoittaa, että jokainen neuroni saa syötteen vain pieneltä joukolta edellisen kerroksen neuroneita – tämä joukko on neuronin *reseptiivinen kenttä* (kuvat 2a ja 2b) [Haykin 2009, 30]. Lisäksi konvoluutioverkko olettaa, että syöte ei ole mikä hyvänsä vektori, vaan että se järjestyy jonkin sisäisen jatkuvuuden perusteella yhteen tai useampaan ulottuvuuteen [Bengio *et al*. 2015, Ch. 9] ja että näissä ulottuvuuksissa lähekkäiset alkiot ovat toisistaan riippuvaisia [LeCun *et al.* 2015]. Konvoluutioverkon neuronin reseptiivinen kenttä onkin siis paikallinen naapurusto: esimerkiksi kuvasyötteessä se olisi pieni ikkuna johonkin kuvan yksityiskohtaan (kuva 2c).



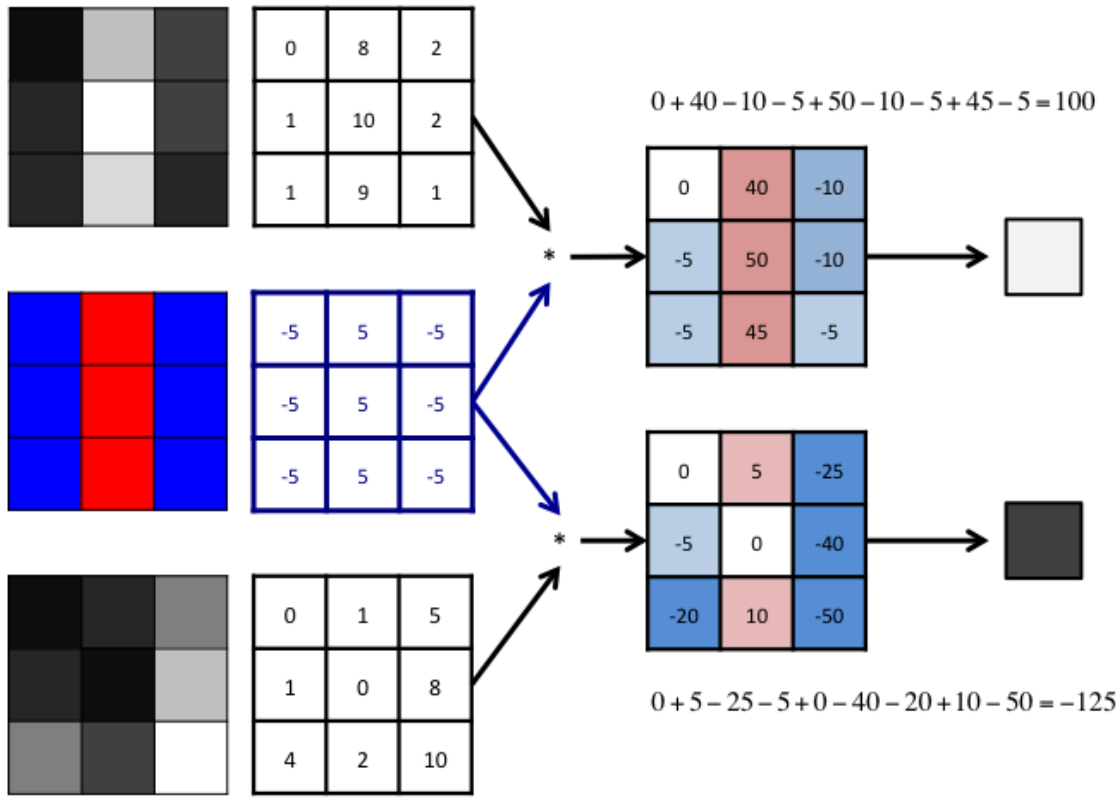
Kuva 2. Täysin yhdistetty (a) ja harvaan yhdistetty (b) neuroverkkokerros. Kuvainformaation (c) tapauksessa reseptiivinen kenttä on kaksiulotteinen.

Tällä paikallisuuden oletuksella on joitakin seurauksia. Perinteiselle FC-verkolle olisi sinänsä aivan sama, vaikka syötekuvat esitettäisiin alkuperäisten pikselivektorien permutaatioina, kunhan muunnos olisi kaikille kuville sama [Bengio *et al*. 2015, Ch. 9.4]. Se oppisi niistä saman minkä permutoimattomista kuvistakin. Konvoluutioverkko sen sijaan ei oppisi: sille suurin osa permutaatioista näyttäytyisi hyvin erilaisina kuin alkuperäinen kuva – kuten ihmisellekin. Konvoluutioverkon oletuksien ulottuvuuksista ja naapuruudesta täytyy siis päteä syötteeseen.

Painokertoimien jakaminen tarkoittaa käytännössä sitä, että kaikki reseptiivisen kenttänsä eri kohdista syötettä saavat neuronit käyttävät samoja painokertoimia eli parametreja. Tällaista parametrisarjaa kutsutaan yleensä *ytimeksi* (kernel), joka kuvainformaation tapauksessa on jälleen vähintään kaksiulotteinen [Bengio *et al*. 2015, Ch. 9.1]. Ydin on ikään kuin suodatin, joka määrittelee ja tunnistaa jotakin tiettyä piirrettä, ja sitä voidaan ajatella ”liu’utettavan” syötteen yli *piirrekartan* (feature map) muodostamiseksi [Haykin 2009, 201]. Intuitiivisesti piirrekarttaa voisi tulkita niin, että se kertoo, missä kohdin syötettä sen etsimää piirrettä esiintyy ja kuinka puhtaana.

Kuva 3 havainnollistaa, miten yksi neuroni laskee tuloksen kahdelle eri syötteelle, joista toinen vastaa sen etsimää piirrettä ja toinen ei. Neuronin yti-

men painoarvot on määritelty välillä -5 ja 5, ja syötearvot välillä 0–10 on tulkittu harmaasävyarvoiksi. Syöte- ja ydinmatriisit kerrotaan vastinalkioittain, ja lopuksi tulot summataan. Kyseessä ei siis ole matriisikertolasku, vaikka sekä syöte että ydin ovat matriiseja. Piirrettä hyvin vastaavalla syötteellä neuroni saa tuloksen 100 ja huonosti vastaavalla -125. Edellinen siis aiheuttaa neuronissa varsin suuren aktivaation ja näkyisi piirrekartassa hyvin vaaleana pisteenä, kun taas jälkimmäinen saisi aikaan tumman pisteen, mikäli normalisoidaan arvot välille 0–10 ja noudatetaan samaa harmaasävytulkintaa kuin syötteessäkin.



Kuva 3. Pystysuoraa viivaa tunnistavan neuronin 3x3 pikselin kokoinen ydin (keskellä). Kynnysarvot on tässä ohitettu.

Edellä havainnollistettu operaatio on diskreetti ristikorrelaatio:

$$f(i,j) = (I * K)(i,j) \sum_m \sum_n I(i+m, j+n) K(m,n),$$

jota käytetään usein laskennallisista syistä varsinaisen konvoluution asemasta koneoppimiskirjastoissa. I on kaksiulotteinen syöte ja K kaksiulotteinen ydin. Yllä annettu kaava määrittelee diskreetin konvoluution, jonka mukaan konvoluutioverkko on nimetty, jos ydin K peilataan siten, että syötteen ensimmäinen arvo kerrotaan ytimen viimeisellä arvolla, syötteen toinen arvo kerrotaan ytimen toiseksi viimeisellä arvolla ja niin edelleen. Ristikorrelaatiota kutsutaan

usein konvoluutioksi, vaikka matemaattisesti kyse on eri käsitteistä. [Bengio *et al*. 2015, Ch. 9.1]

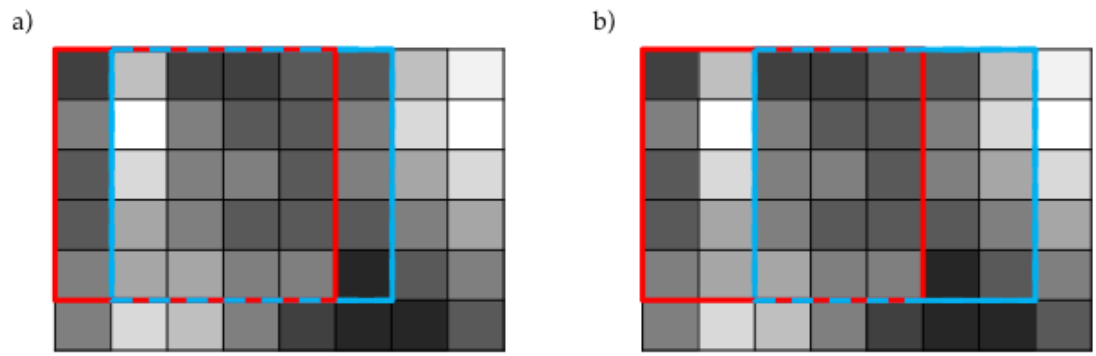
Painokertoimien jakaminen sisältää sen oletuksen, että jos jokin piirre on tärkeä, se on tärkeä missä kohtaa syötettä tahansa, ja se on myös joka paikassa samanlainen [Bengio *et al*. 2015, Ch. 9.2]. Tyypillisiä ensimmäisen konvoluutiokerroksen tunnistamia piirteitä ovat esimerkiksi reunat: pätkä pystysuoraa reunaa on olennaisesti ottaen samanlainen missä hyvänsä päin kuvaa se esiintyykin. Yleensä tämä ominaisuus on kuviontunnistuksessa etu, mutta poikkeuksiakin tietysti löytyy [Bengio *et al*. 2015, Ch. 9.2].

Edellä mainitut rajoitukset – harva yhdistäminen ja painokertoimien jakaminen – vähentävät opittavien parametrien määrää huomattavasti puuttumatta kuitenkaan lainkaan neuronien määrään. Seuraavaksi esitellään kaksi menetelmää, joilla neuroneita saadaan vähennettyä.

3.3. Neuronien vähentäminen

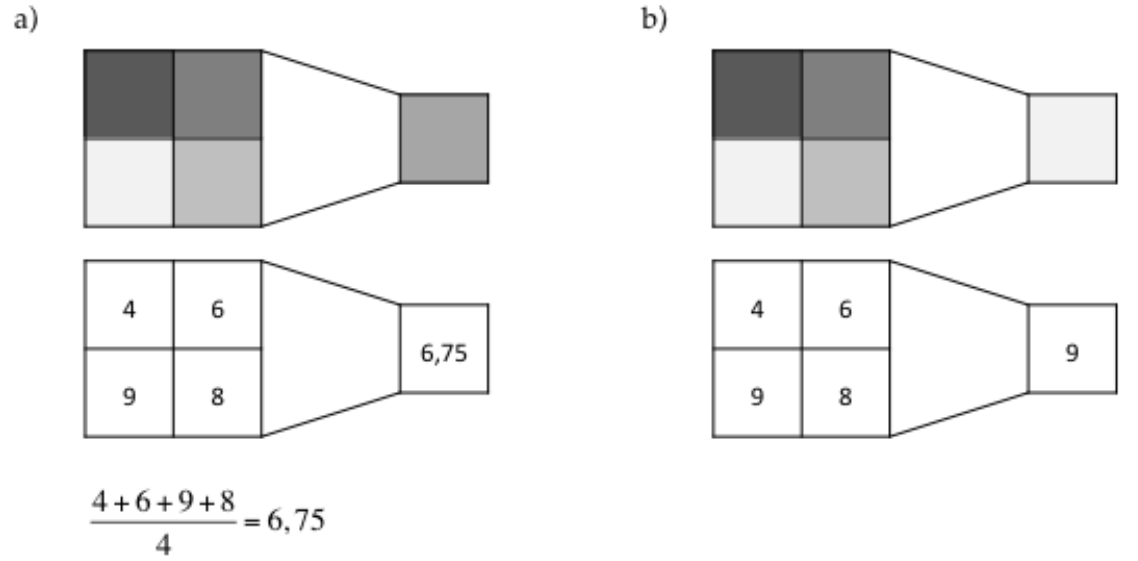
Konvoluutioverkossa neuroneiden määrää vähennetään tyypillisesti *askelluksen* (stride) ja *kokoomakerrosten* (pooling layer) avulla. Ensimmäinen toimii konvoluutiokerroksen yhteydessä, kun taas jälkimmäinen mielletään yleensä omaksi erilliseksi kerroksekseen.

Askellus tarkoittaa pikselimäärää tai muuta syötteen yksikköä, jonka verran ydintä siirretään vierekkäisten neuronien välillä konvoluutiokerroksessa (kuva 4). Reunojen yli menevät reseptiiviset kentät ovat mahdollisia, kun kuva ympäröidään nolla-arvoilla (padding). Jos askellus on yksi, neuroneita on konvoluutiokerroksessa yhtä paljon kuin pikseleitä syötteessä. Askellus on kuitenkin joskus järkevää säätää suuremmaksi: esimerkiksi arvolla kaksi neuronien määrä putoaa heti neljännekseen, mikäli syöteulottuvuuksia on kaksi ja molempien askellus on sama. Suurempaa askellusta käytetäänkin etenkin verkon ensimmäisessä kerroksessa, jossa sen vaikutus muistin tarpeeseen on varsin merkittävä. [Karpathy 2015]



Kuva 4. 5x5-kokoisen ytimen askellus arvolla 1 (a) ja 2 (b).

Kokoomakerrokset puolestaan ovat varsin yksinkertaisia välikerroksia, joiden tehtävä on pienentää resoluutiota edustamalla tietyn alueen aktivaatiota jollakin tunnusluvulla, esimerkiksi keskiarvolla tai suurimmalla arvolla (max-pooling) (kuva 5). Kokoomakerrokset paitsi vähentävät neuronien määrää niin myös tekevät tunnistuksesta invariantin pienille siirtymille ja epäjatkuvuuksille [Bengio *et al*. 2015, Ch. 9.3]. Samalla pikselintarkka tieto piirteen sijainnista tietenkin kadotetaan, mikä ei aina ole tarkoituksenmukaista [Bengio *et al*. 2015, Ch. 9.4].



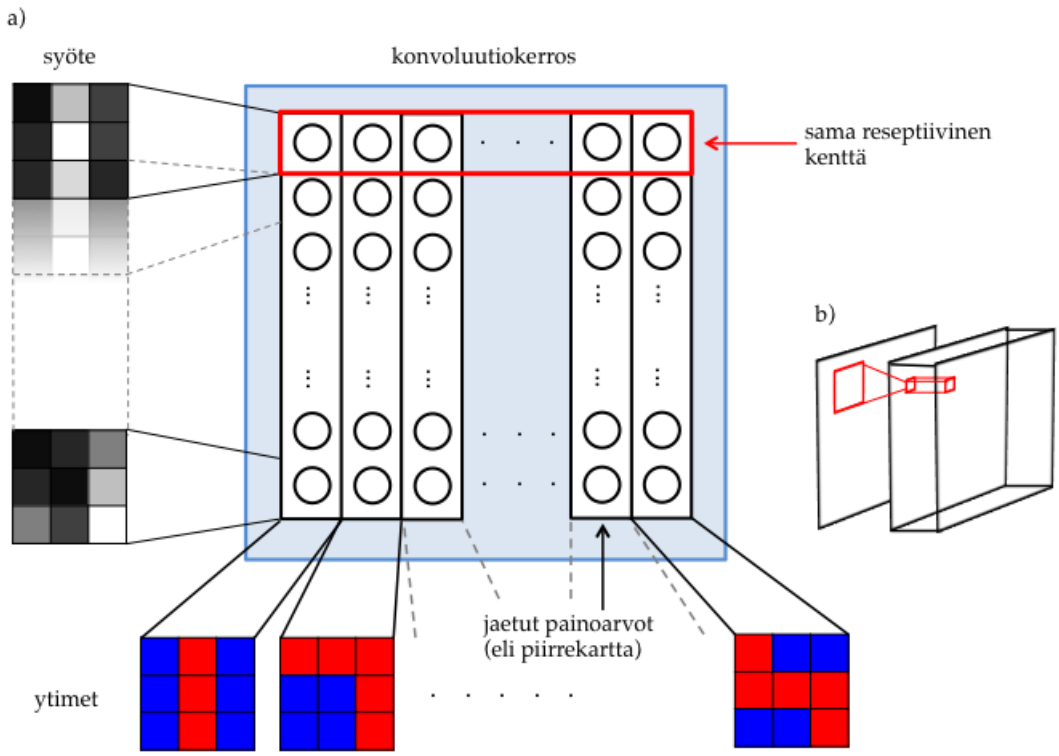
Kuva 5. Kokoomaneuronin toiminta keskiarvolla (a) ja suurimmalla arvolla (b), kun reseptiivisen kentän koko on 2x2 pikseliä.

Kokoomakerroksen neuronien reseptiivisen kentän koko on yleensä 2x2 tai 3x3 pikseliä, ja askellus voi olla saman verran tai vähemmän, jolloin reseptiiviset kentät menevät hieman päällekkäin, kuten konvoluutiokerroksissakin [Karpathy 2015].

3.4. Kokonaisarkkitehtuuri ja opetus

Edellä kuvailtiin tilannetta, jossa ydin suodattaa syötettä halutun piirteen löytämiseksi. Yksi ydin voi kuitenkin suodattaa eli tunnistaa vain yhtä piirrettä, joten käytännössä suodattimia täytyy yhdessä konvoluutiokerroksessa olla aina useita, jopa useita satoja [Krizhevsky *et al*. 2012]. Kaksiulotteista syötettä käsittelevän konvoluutiokerroksen voi siis ajatella kuvan 6 mukaisesti kolmiulotteisena rakenteena: sen pituus ja leveys vastaavat syötteen pituutta ja leveyttä, ja sen jokainen syvyystaso vastaa yhtä suodatinta. Syvyyssuunnassa päällekkäin sijaitsevat neuronit katselevat samaa reseptiivistä kenttää, kun taas vierekkäin samalla tasolla olevat jakavat saman ytimen eli samat painokertoimet [Karpat-

hy 2015]. Suodatuksen tuloksena jokainen syvyystaso muodostaa oman piirrekarttansa, joka voidaan hahmottaa aktivaation suuruutta syötteen eri osissa esittävänä kuvana. Nämä kuvat puolestaan annetaan syötteeksi seuraavalle konvoluutiokerrokselle, joka käsittelee niitä aivan samoin kuin se käsittelisi varsinaista kuvasyötettäkin.

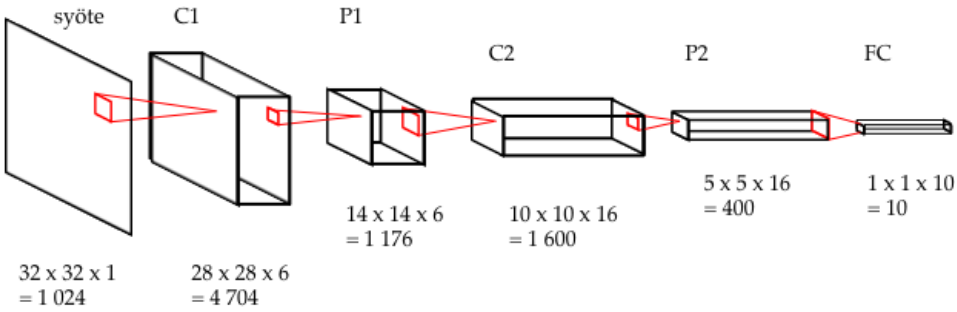


Kuva 6. (a) Jaetut painot ja reseptiiviset kentät, kaksiulotteinen yksinkertaistus. (b) Reseptiivinen kenttä kolmiulotteissa konvoluutiokerroksessa.

Konvoluutioverkon kokonaisarkkitehtuuri yhdistelee yleensä konvoluutiokerroksia (C), kokoomakerroksia (P) sekä perinteisiä, täysin yhdistettyjä FC-kerroksia [Karpathy 2015]. Toisinaan myös epälineaarisuuden laskenta eli aktivaatiofunktio mielletään omaksi kerroksekseen (A). Joissakin lähteissä koko C-A-P-rakennetta kutsutaan yhden konvoluutiokerroksen eri vaiheiksi [Bengio *et al.* 2015, Ch. 9.3], mutta tässä työssä niistä puhutaan terminologian yksinkertaistamiseksi omina erillisinä kerroksinaan.

Kokonaisarkkitehtuurin hahmottamiseksi otetaan tarkasteluun yksinkertaistettu esimerkki, joka pohjautuu löyhästi LeCunin ja kumppaneiden LeNet-5 verkkoon [LeCun *et al.* 1998]. Kuva 7 antaa yleiskuvan rakenteesta. Verkon syöte on harmaasävykuva, jonka koko on 32x32 pikseliä, ja sen tuloste on kymmenenluokkainen luokittelu: LeNet-verkot suunniteltiin tunnistamaan käsinkirjoitettuja numeroita 0–9. Esimerkkiverkkomme tosin olisi luultavasti liian yksinkertainen selviytymään tästä tehtävästä. Verkossa on kaksi C-kerrosta, kaksi P-

kerrosta ja yksi FC-kerros, sekä jokaista C-kerrosta seuraava epälineaarinen aktivaatiofunktio, jota ei kuitenkaan kuvaan ole erikseen piirretty. Ensimmäinen C-kerros sisältää 6 eri piirrekarttaa ja toinen 16, ja molemmat P-kerrokset pudottavat piirrekarttojen resoluution neljännekseen. Resoluution ja syvyyden tulo on kerroksen neuroneiden määrä. FC- ja C-kerrosten neuronien reseptiivisen kentän koko on 5x5 pikseliä ja askellus 1, ja P-kerrosten 2x2 pikseliä ja askellus 2.



Kuva 7. Yksinkertainen konvoluutioarkkitehtuuri, LeNet-5 verkkoa mukaillen [LeCun *et al*. 1998].

Tarkastellaan vielä verkkoa lukuina (taulukko 1). Kerroksen neuronien määrä saadaan pituuden, leveyden ja syvyyden tulona. FC- ja C-kerrosten neuronit on yhdistetty edellisen kerroksen kaikkiin syvyystasoihin eli piirrekarttoihin, P-kerrokset puolestaan ovat piirrekarttakohtaisia. Reseptiivinen kenttä on samalla kerroksen ytimen koko, ja kun siihen lisätään yksi kynnysarvo ja summa kerrotaan suodattimien määrällä, saadaan kerroksen opittavien parametrien määrä. Parametrien kokonaismäärä saadaan pituuden, leveyden ja opittavien parametrien tulona. Nyt huomataan, miten paljon painokertoimien jakaminen vähentää verkon vapausasteita: ilman jakamista kaikki parametrit olisi opittava erikseen. Mikäli kerrokset eivät lisäksi olisi harvaan yhdistettyjä, parametrien määrä olisi vielä monta kertaluokkaa suurempi: esimerkiksi C1-kerroksen tapauksessa lähes 5 miljoonaa (1024 x 4704 + kynnysarvot). P-kerroksen neuronit eivät tässä ratkaisussa ole oppivia.

	pituus x leveys	syvyys	neuro-neita	reseptii-vinen kenttä	kynnys-arvoja	opittavia para-metreja	para-metreja yhteensä
syöte	32 x 32	1	1 024	-	-	-	-
C1	28 x 28	6	4 704	5 x 5 x 1	6	156	122 304
P1	14 x 14	6	1 176	2 x 2 x 1	0	0	0
C2	10 x 10	16	1 600	5 x 5 x 6	16	2 416	241 600
P2	5 x 5	16	400	2 x 2 x 1	0	0	0
FC	1 x 1	10	10	5 x 5 x 16	10	4 010	4 010
yht.	-	-	8 914	-	-	6 582	367 914

Taulukko 1. Esimerkkiverkon rakenne lukuina.

Konvoluutioverkoille on tyypillistä, että suurin osa neuroneista sijaitsee ensimmäisissä kerroksissa ja ylempiä kerroksia kohden niiden määrä tasaisesti vähenee, kun taas suurin osa opittavista parametreista sijaitsee viimeisillä FC-kerroksilla – etenkin niiden välissä, mikäli FC-kerroksia on useampi. Suurimmat muistivaatimukset opetusvaiheessa sijoittuvat siis alimmille kerroksille, kun neuronien aktivaatioita täytyy tallentaa. Sen takia suuremman askelluksen käyttö näissä kerroksissa on usein perusteltua. [Karpathy 2015]

Konvoluutioverkot opetetaan perinteiseen tapaan *takaisinlevitysalgoritmilla* (backpropagation), joka optimointimenetelmineen on niin laaja aihe, että se jätetään tässä yhteydessä käsittelemättä. Perustiedot takaisinlevitysalgoritmista löytyvät alan kursseista ja perusteoksista [Karpathy 2015; Juhola 2015; Haykin 2009], ja Bengio [2012] kokoaa yhteen paljon käytännön tietoa ja teoriaa edistyneemmistä optimointimenetelmistä, joilla on merkitystä syvien verkkojen opettamisessa. Mitkään näistä menetelmistä eivät kuitenkaan ole erityisesti konvoluutioverkoille suunniteltuja, vaan pätevät kaikenlaisille syville neuroverkoille.

Syväoppimisen alkuaikoina *ohjaamattomalla esiopetuksella* (unsupervised pretraining) oli huomattava rooli syvien verkkojen opetuksessa, mutta sittemmin luokiteltujen aineistojen kasvaessa sen rooli on pienentynyt [LeCun *et al*. 2015]. Nykyiset konvoluutioverkot opetetaankin usein alusta loppuun ohjatusti. Opetusjoukon läpikäynnissä hyödynnetään usein *stokastista gradientin laskeutumista* (stochastic gradient descent) eli jokaisen takaisinlevityksen välissä ei käydä läpi koko opetusjoukkoa, vaan vain jokin sen satunnainen osajoukko. Menetelmä tuottaa varsin hyviä oppimistuloksia suhteellisen nopeasti, joten siitä on tullut varsin suosittu [LeCun *et al*. 2015; Bengio 2012, 5–6].

Yleisesti ottaen syvien neuroverkkojen opettamista vaivaa nk. katoavien tai räjähtävien gradienttien ongelma (vanishing/exploding gradients) [Schmidhuber 2015, 93–94], jota on yritetty helpottaa mm. ohjaamattomalla esiopetuksella [Bengio 2009, 32] ja uudenlaisilla aktivaatiofunktioilla, kuten ReLU [Bengio *et al*. 2015, Ch. 6.3.1]. Konvoluutioverkkojen kohdalla gradientit eivät kuitenkaan ole niin suuri ongelma – mahdollisesti niiden rakenteellisten ominaisuuksien takia [Bengio 2009, 24].

4. Kuvantunnistus konvoluutioverkolla

Tässä luvussa tarkastellaan, miten konvoluutioverkot suoriutuvat kuvantunnistustehtävästä, erityisesti kuvanluokittelusta. Aluksi tutustutaan konvoluutioverkkojen historiaan ja kehitykseen luokittelijoina, sitten pohditaan niiden käyttämiä representaatioita ja lopuksi etsitään syitä niiden menestykselle.

4.1. Konvoluutioverkkojen läpimurto

Ensimmäisinä konvoluutioverkkoina pidetään hieman lähteestä riippuen joko Fukushiman [1980] Neocognitronia tai LeCunin ja hänen ryhmänsä [1989] LeNet-verkkoja. Molemmat suunniteltiin nimenomaisesti tunnistamaan visuaalisia hahmoja, ja ainakin Neocognitron sai inspiraationsa suoraan Hubelin ja Wieselin klassisesta apinan näköjärjestelmän hierarkiamallista [Fukushima, 1980]. Konvoluutioverkoilla ja kuvantunnistuksella on siis pitkä yhteinen historia, mutta siitä huolimatta konvoluutioverkot olivat syrjässä konenäön kehityksen valtavirrasta monta vuosikymmentä [Jones 2014]. Poikkeuksen tähän muodostivat shekkien automaattiseen lukemiseen käytetyt verkot, jotka LeCun ryhmineen kehitti ja jotka tulivat yleiseen käyttöön jo 1990-luvulla [LeCun *et al*. 1998] sekä muutamat muut sovellukset [LeCun *et al*. 2015].

Konvoluutioverkot tekivät varsinaisen läpimurtonsa kuvantunnistuksen valtavirtaan vuonna 2012, kun Krizhevskyn ja ryhmän [2012] AlexNet päihitti kilpailijansa ILSVRC:n molemmissa silloisissa kilpailukategorioissa (kuvanluokittelu ja yhden kohteen paikannus) reilulla marginaalilla. Esimerkiksi kuvanluokittelun kohdalla edellisenä vuonna voittoon oli riittänyt 25,8%:n virhe, minkä AlexNet pudotti kertaheitolla 16,4%:iin tehden selvän eron toiseksi tulleen järjestelmän 26,2%:iin [Russakovsky *et al*. 2015]. Siitä asti ILSVRC:ssä ovatkin kisanneet lähes pelkästään erilaiset konvoluutioverkoille perustuvat järjestelmät [Russakovsky *et al*. 2015]. Viimeisimmän ennätyksen kuvanluokittelussa teki Googlen tutkimusryhmän GoogLeNet [Szegedy *et al*. 2014] vuonna 2014 6,7%:llaan [Russakovsky *et al*. 2015]. Ei siis ole liioittelua sanoa, että ILSVRC:n puitteissa tapahtunut kehitys on ollut lähes yksinomaan konvoluutioverkkojen ansiota.

Edellisen luvun esimerkkiin verrattuna AlexNet [Krizhevsky *et al.* 2012] on rakenteeltaan jo huomattavasti monimutkaisempi. Se pitää sisällään viisi C-kerrosta, kolme P-kerrosta, kolme FC-kerrosta sekä monenlaisia oppimista auttavia tekniikoita, joita kaikkia ei tässä ole mielekästä referoida. Yleisesti ottaen verkko oli 60 miljoonalla parametrillaan opetusjoukon suuruudesta huolimatta (1,2 miljoonaa kuvaa) taipuvainen ylioppimaan, joten sen opettaminen vaati monenlaisia aputekniikoita. Oppimista autettiin mm. laajentamalla opetusjoukkoa erilaisin luokituksen säilyttävin muunnoksin (eri rajauksia, värien ja valoisuuden variaatioita) sekä niin kutsutulla tiputusmenetelmällä (dropout), jota sovellettiin FC-kerroksiin oppimisen aikana [Krizhevsky *et al.* 2012]. Kuten aiemmin todettiin, suurin osa konvoluutioverkon opittavista parametreista sijaitsee näillä kerroksilla, ja ne ovat siksi ensisijainen syyllinen verkon ylioppimiseen [Lin *et al*. 2014]. Merkittävä edistysaskel oli myös prosessointitehtävän jakaminen kahdelle grafiikkaprosessorille, mikä nopeutti opettamista. Aineiston esiprosessointia sen sijaan ei käytetty juuri lainkaan: verkko opetettiin suoraan opetusjoukon yli keskitetyillä, normalisoimattomilla RGB-arvoilla [Krizhevsky *et al.* 2012], mikä eroaa huomattavasti perinteisemmästä konenäkölähestymisestä (esimerkiksi Davies 2005, 5–6).

GoogLeNetin [Szegedy *et al*. 2014] arkkitehtuuri on liian mutkikas missään kattavassa mielessä selitettäväksi, mutta yleisesti ottaen se hyödyntää ja jalostaa Linin ja kumppaneiden [2014] ajatusta ”verkoista verkon sisällä” (network-in-network). GoogLeNetin nk. Inception-moduulit koostuvat suodattimista, joiden ytimet ovat eri kokoisia. Näin verkko pystyy ikään kuin tarkastelemaan kuvan piirteitä useassa eri mittakaavassa samaan aikaan ja saman kerroksen tasolla – toisin kuin perinteinen konvoluutioverkko, jonka reseptiiviset kentät laajenevat suoraviivaisesti kerroksesta toiseen. Inception-moduuleiden lisäksi verkossa on tavallisia C-, P- ja FC-kerroksia, kaikkiaan 22 oppivaa kerrosta [Szegedy *et al*. 2014].

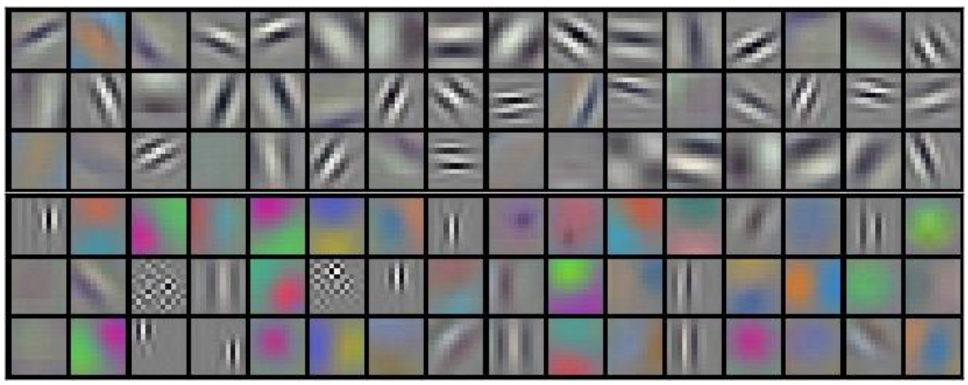
4.2. Representaatioista

Kuvantunnistusta ja luokittelua on perinteisesti lähestytty kaksi- tai kolmiportaisena rakenteena, jossa erilaisten suodatusten ja piirteiden irrotuksen jälkeen sovelletaan mahdollisesti joitakin välitason menetelmiä, ja viimeistelyn hoitaa oppiva luokittelija [Davies 2005, 6]. Konvoluutioverkossa nämä kaikki vaihet yhdistyvät yhdeksi, alusta loppuun asti oppivaksi järjestelmäksi – joskin viimeisten FC-kerroksien voi ajatella toimivan enemmän luokittelijan roolissa, kun taas alemmat konvoluutiokerrokset keskittyvät piirteiden irrottamiseen. Konvoluutioverkko on representaation oppimisen menetelmä: se ei saa mitään

etukäteistietoa tärkeistä piirteistä, vaan päättelee ne opetuksen aikana itse ja muodostaa niistä oman representaationsa.

Tässä mielessä onkin kiinnostavaa, millaisia piirteitä ja representaatioita kuvia tunnistavat konvoluutioverkot muodostavat. Ikävä kyllä neuroverkkojen hajautetut representaatiot eivät ole samassa mielessä suoraan tarkasteltavissa kuin perinteisempien järjestelmien, joten joudumme tyytymään muutamaan suuntaa-antavaan ja jopa hieman intuitiiviseen kuvaukseen. Läpikulkevana periaatteena on kuitenkin järjestelmän voimakas hierarkkisuus: jokaisella kerroksella on oma käsittelyn tasonsa, ja siten myös omat representaationsa.

Ensimmäisenä esimerkkinä tarkastellaan AlexNetin [Krizhevsky *et al*. 2012] alimman konvoluutiokerroksen oppimia 11x11x3-kokoisia ytimiä, joita on kaikkiaan 96 kappaletta (kuva 8). Kuvat siis edustavat piirteitä, joilla kukin suodatin aktivoituu maksimaalisesti. Osa niistä muistuttaa huomattavan paljon nk. *Gabor-suodattimia* (Gabor filters) [Fisher *et al*. 2014, 106], joita on menestyksekkäästi käytetty käsinmääriteltyinä piirteinä monenlaisissa hahmontunnistusjärjestelmissä [Krüger *et al*. 2013]. Tässä tapauksessa ne on siis kuitenkin opittu suoraan ILSVRC:n opetusjoukon kuvista. Gabor-aallokkeiden on todistettu koodaavan luonnollista kuva-informaatiota varsin tehokkaasti, ja myös kädellisten primääri näköaivokuori hyödyntää samantyyppisiä representaatioita [Krüger *et al*. 2013].

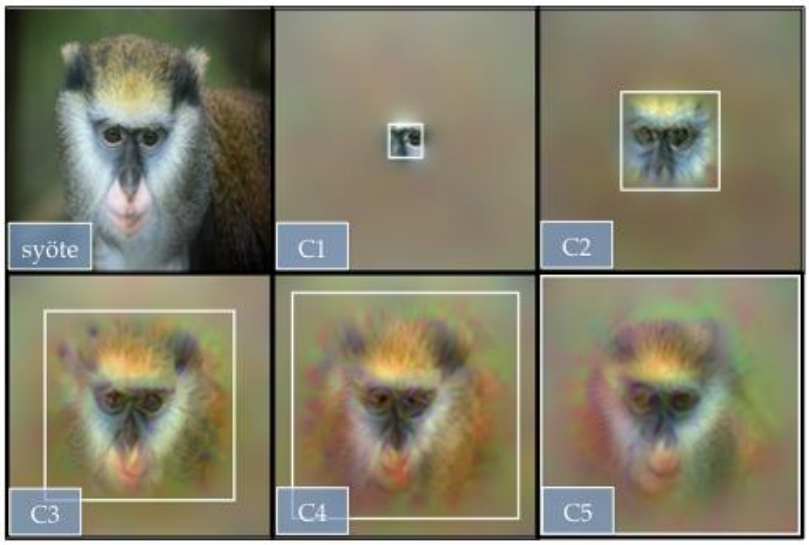


Kuva 8. AlexNetin ensimmäisen kerroksen oppimat ytimet, jotka jakautuvat väriherkkiin ja värisokeisiin suodattimiin, kuten biologinenkin järjestelmä [Krüger *et al*. 2013]. Kuvan lähde Krizhevsky *et al.* [2012].

Mahendran ja Vedaldi [2014] tutkivat AlexNet-arkkitehtuurin piilokerrosten toimintaa. Heidän käyttämänsä verkko oli hyvin pitkälti sama kuin Krizhevskyn ja työryhmän [2012], ja se oli opetettu samalla ImageNet-aineistolla. Heidän menetelmänsä kääntää verkon tunnistavasta generatiiviseksi järjestelmäksi ottamalla kultakin kerrokselta talteen sen syötekuvasta muodostaman representaation ja yrittämällä sen perusteella rekonstruoida alkuperäisen ku-

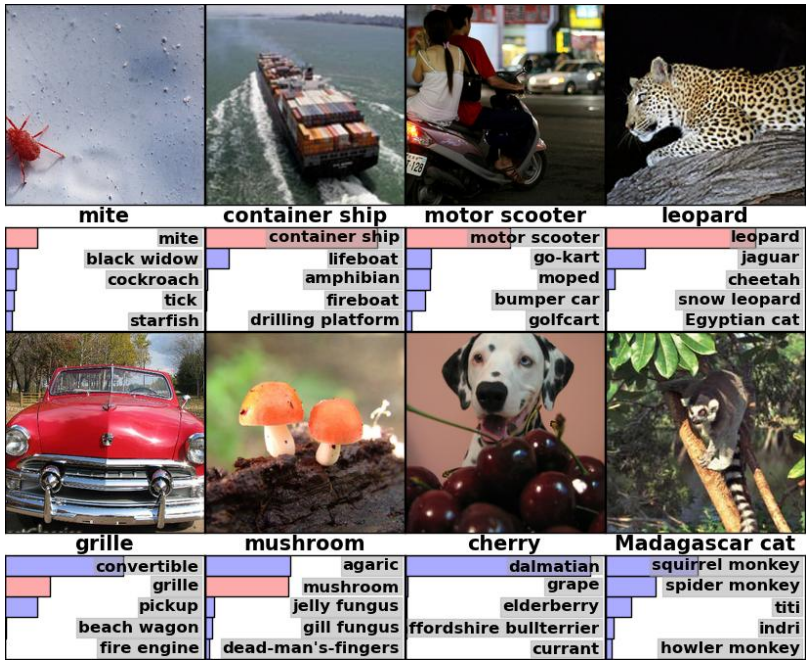
van. Koska konvoluutioverkon perusluonteeseen kuuluu epäolennaisen informaation karsiminen, rekonstruktio ei tietenkään ole uniikki, vaan vain yksi mahdollinen verkon mielestä alkuperäistä kuvaa vastaava esimerkki. [Mahendran and Vedaldi 2014]

Tarkastellaan esimerkiksi konvoluutiokerrosten keskimmäistä 5x5-neuronin joukkoa (kuva 9). Kun Mahendran ja Vedaldi rekonstruoivat niistä syötekuvan, he saivat näkyviin sen, miten korkeampien konvoluutiokerrosten reseptiiviset kentät suhteessa alkuperäiseen syötteeseen laajenevat kerros kerrokselta. Ilmiö on seurausta verkon hierarkkisesta rakenteesta ja resoluution asteittaisesta pienentämisestä: jokainen käsittelykerros kokoaa yhteen informaatiota suuremmalta ja suuremmalta alueelta, kunnes lopulta viimeiset kerrokset ottavat huomioon koko syötealan. Ollaan siis päädytty tilanteeseen, jossa jokainen syötteen pikseli on epäsuorasti yhdistetty jokaiseen tuloskerroksen neuroniin. Laajenevat reseptiiviset kentät ovat myös nisäkkäiden näköjärjestelmän keskeinen ominaisuus [Krüger *et al*. 2013].



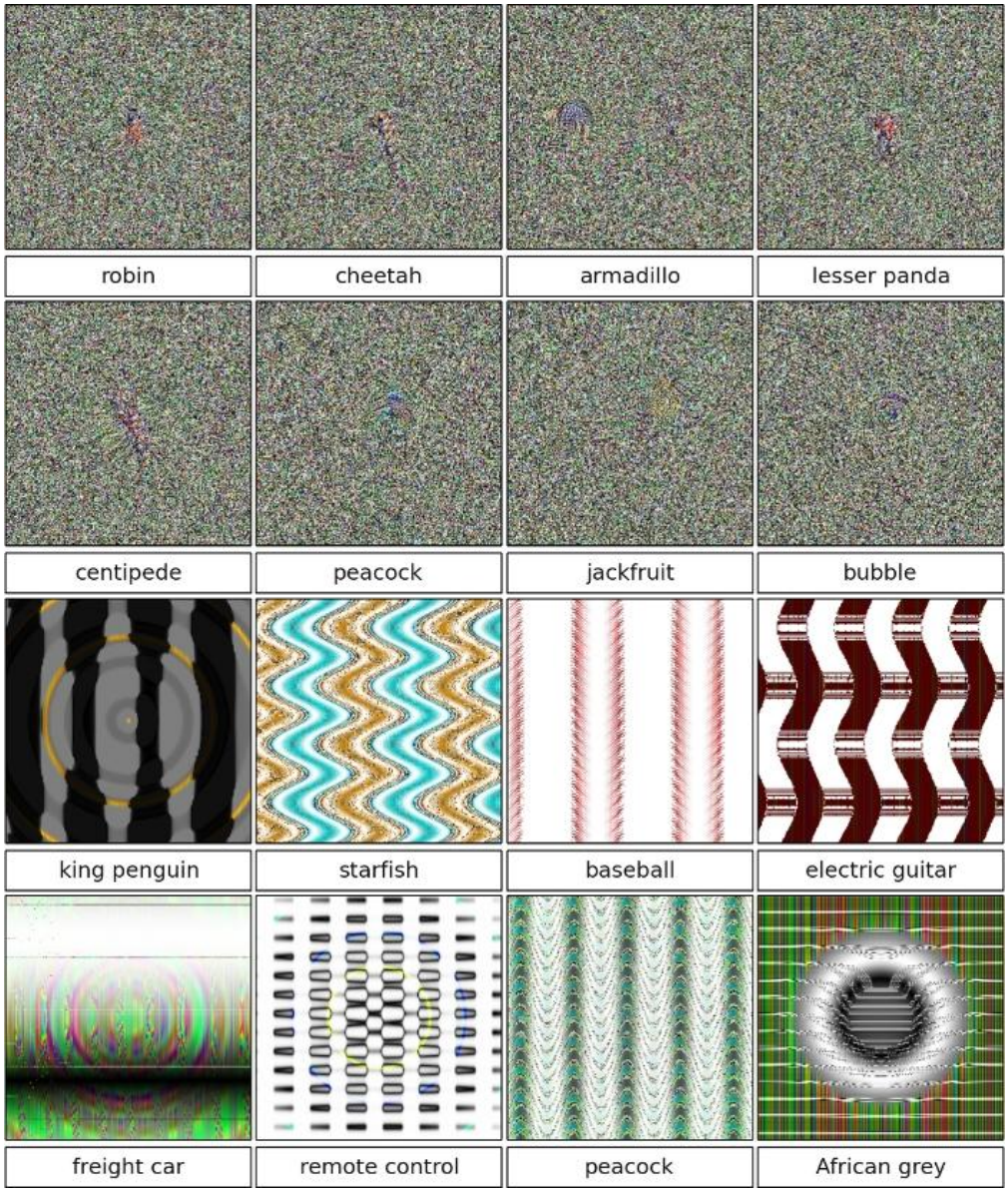
Kuva 9. Kuvasyötteen rekonstruktio ja laajenevat reseptiiviset kentät (valkoinen neliö). Konvoluutiokerrosten jälkeiset FC-kerrokset kattavat koko kuva-alan. Kuvat Mahendran ja Vedaldi [2014].

Tarkastellaan lopuksi vielä joitakin esimerkkejä luokituksista. Kuvassa 10 on AlexNetin tuottamia luokituksia ILSVRC 2010 testiaineistolla sekä kuvien oikeat luokitukset [Krizhevsky *et al*. 2012]. Kuvista saa käsityksen ImageNet-aineiston haastavuudesta ja luokitustehtävän hienojakoisuudesta: kaikki verkon ehdottamat luokat ovat tehtävässä olemassa olevia luokkia. Verkko vaikuttaa kehittäneen varsin hyvän ymmärryksen kuvista, ja sen virheetkin tuntuvat melko ymmärrettäviltä.



Kuva 10. AlexNetin arvioita eräistä ILSVRC 2010 -testikuvista. Palkkien pituus kuvaa arvion varmuutta. Kuva Krizhevsky *et al.* [2012].

Konvoluutioverkko ei kuitenkaan ole ihmisnäkö, vaan se on helposti huijattavissa. Nguyen *et al.* [2014] loivat AlexNetille optimoituja kuvia, jotka se tunnistaa 99,9%:n varmuudella johonkin luokkaan kuuluvaksi mutta joissa ei ole mitään ihmissilmälle tunnistettavaa (kuva 11). Esimerkit ovat paitsi hupaisia myös informatiivisia: niiden valossa vaikuttaa epätodennäköiseltä, että verkon käyttämä representaatio millään lailla mallintaisi kohteiden kolmiulotteista muotoa. Ainakaan AlexNetin tapauksessa kolmiulotteista analyysia ei siis tarvita tunnistuksen onnistumiseen, mutta tämä puute epäilemättä rajaa sen käyttökelpoisuutta muunlaisissa konenäön tehtävissä.



Kuva 11. AlexNetille optimoituja huijauskuvia. Ylemmissä kuvissa pikselit ovat saaneet muokkautua toisistaan riippumatta, alemmissa niitä on rajoitettu luonnollisemman tuloksen aikaansaamiseksi. Kuvat Nguyen *et al.* [2014].

4.3. Miksi konvoluutioverkko toimii

Puutteistaan huolimatta syvät konvoluutioverkot edustavat tällä hetkellä ehdotonta kuvantunnistuksen huippua [LeCun *et al*. 2015]. Mikä tekee niistä niin paljon aiempia, matalia arkkitehtuureja toimivampia?

Biologinen näköjärjestelmä on jopa monimutkaisten aivojemme mittakaavassa poikkeuksellisen syvä hierarkia, mikä luultavasti heijastelee näkötehtävän rakennetta: visuaalinen maailma koostunee luonnostaan piirteistä, jotka liittyvät yhteen muodostaen askel askeleelta monimutkaisempia rakenteita [Krüger *et al*. 2013]. On tehokkaampaa niin oppimisen kuin tiedonkäsittelynkin

kannalta käyttää sellaista representaatioiden hierarkiaa, joka hyödyntää tätä tehtävän ominaisuutta [Krüger *et al*. 2013].

Asiassa on myös laskennallinen puolensa: Bengion [2009, 14] mukaan ainakin jotkin funktiot, joka ovat tehokkaasti representoitavissa arkkitehtuurilla, jonka syvyys on k, vaativat eksponentiaalisen määrän elementtejä mikäli arkkitehtuurin syvyys on k–1. Neuroverkkojen tapauksessa se tarkoittaisi eksponentiaalista lisäystä opittavien parametrien määrään, mikä on valtava ongelma paitsi ajan- ja muistinkäytön niin myös ylioppimisen kannalta. On toki totta, että mikä hyvänsä syvä neuroverkko on periaatteessa esitettävissä tarpeeksi "leveällä" kolmikerroksisella verkolla (Kolmogorovin teoreeman sovellus), mutta se ei silti välttämättä ole käytännössä mahdollista [Bishop 1995, 137–140] tai ainakaan laskennan näkökulmasta tehokasta [Bengio 2009, 18–20]. Syvien verkkojen käytölle on siis ainakin joissakin tapauksissa hyvät perusteet, ja kuvantunnistusongelma vaikuttaisi olevan yksi niistä.

Syvyyden ja hierarkkisuuden lisäksi konvoluutioverkkoja luonnehtivat niiden rakenteen yksityiskohdat. Paikalliset ja asteittain laajenevat reseptiiviset kentät sekä neuronien järjestäytyminen avaruudellisen läheisyyden perusteella (retinotopic organisation) ovat biologisen näön keskeisimpiä ominaisuuksia, etenkin käsittelyn varhaisissa vaiheissa [Krüger *et al*. 2013]. Konvoluutioverkko on suunniteltu noudattamaan samoja periaatteita, ja myös kokoomakerrosten käytön taustalla on biologinen esikuva [Bengio *et al*. 2015, Ch. 9.10]. Konvoluutioverkko ei toki ole biologisen järjestelmän suora kopio, mutta se on ammentanut siitä huomattavan paljon inspiraatiota, mikä on varmasti osaltaan vaikuttanut sen menestykseen kuvantunnistuksessa.

Konvoluutioverkko ei kuitenkaan ole yleinen konenäköongelman ratkaisija: kuvan analysoinnin kentällä on monia tehtäviä, joihin se ei sovellu. Se on ensisijaisesti luokittelija ja hahmontunnistaja, jonka ei esimerkiksi voi odottaa ratkaisevan toiminnan ohjaamiseen tarvittavan kolmiulotteisen representaation ongelmaa.

5. Yhteenveto

Tässä työssä on esitelty konvoluutioverkkojen käyttöä kuvantunnistuksessa ja erityisesti kuvanluokittelutehtävässä. Konvoluutioverkot ovat päihittäneet kirkkaasti muut menetelmät ImageNet-kuva-aineistoon perustuvassa ILSVRC–kisassa ja asettaneet alalle uudet standardit. Niiden menestys perustuu luultavasti laskennan rajoittamiseen melko yleisten mutta kuitenkin tehokkaiden luonnollisten rajoitusten avulla, sekä niiden rakenteen syvyyteen ja hierarkkisuuteen, joka sopii tehtävän luonteeseen.

Tietenkin myös teholla ja aineistolla on merkitystä: on selvää, ettei tämänhetkisiä huippusuorituksia olisi syntynyt ilman tehokkaampia prosessoreita, rinnakkaista prosessointia ja massiivisia opetusaineistoja, kuten ImageNet [Anthes 2013]. Toisaalta konvoluutioverkot päihittivät kilpailijansa pienimuotoisessa kuvantunnistuksessa (MNIST) jo 1990-luvulla [LeCun *et al*. 1998], joten niiden menestyksen takana on muutakin kuin vain lisääntynyt laskentateho. Jää nähtäväksi, miten konvoluutioverkkopohjaiset tunnistusjärjestelmät tulevat hyötymään aidosti rinnakkaista laskentaa käyttävistä mikropiireistä, joita parhaillaan monilla tahoilla kehitetään [LeCun *et al*. 2015].

Algoritmisesti ottaen tulevaisuuden kannalta erityisen kiinnostavia ovat menetelmät, joilla voidaan auttaa verkkoa keskittymään syötteen olennaisimpiin osiin. Biologinen näköjärjestelmä vaatii visuaalista huomiota (visual attention) rajaamaan laskentaansa [Farah 2000, 174], sillä ilman sen ohjausta yleistetty näköongelma on kerta kaikkiaan liian vaikea ratkaistavaksi [Tsotsos 1990]. Konvoluutioverkko, sellaisena kuin se on tässä työssä esitetty, ei kuitenkaan yksinään pysty simuloimaan visuaalista huomiota, koska yksi sen keskeisiä toimintaperiaatteita on piirteiden kohteleminen samanarvoisina syötteen joka kohdassa, mikä on seurausta jaetuista painoarvoista. Siksi tarvitaan jonkinlaista menetelmien yhdistelmää, ja eräs tulevaisuuden mahdollisuus onkin konvoluutioverkon, takaisinsyöttävän verkon ja vahvistusoppimisen (reinforcement learning) yhdistäminen [LeCun *et al*. 2015].

6. Lähteet

Gary Anthes. 2013. Deep learning comes of age. *Comm. ACM* 56, 6, 13-15.

Yoshua Bengio. 2009. *Learning Deep Architectures for Artificial Intelligence*. Foundations and Trends in Machine Learning, Now Publishers.

Yoshua Bengio. 2012. Practical Recommendations for Gradient-Based Training of Deep Architectures. arXiv: 1206.5533v2.

Yoshua Bengio, Ian J. Goodfellow and Aaron Courville. 2015. *Deep Learning*. Book in preparation for MIT Press. http://www.deeplearningbook.org. Checked 17.2.2016.

Christopher M. Bishop. 1995. *Neural Networks for Pattern Recognition*. Clarendon Press.

Edward Roy Davies. 2005. *Machine Vision: Theory, Algorithms, Practicalities (3rd ed)*. Morgan Kaufmann Publishers.

Mark Everingham, S. M. Ali Eslami, Luc Van Gool, Christopher K. I. Williams, John Winn, Andrew Zisserman. 2015. The PASCAL Visual Object Classes Challenge: A Retrospective. *International Journal of Computer Vision* 111, 98–136.

Robert B. Fisher, Toby P. Breckon, Kenneth Dawson-Howe, Andrew Fitzgibbon, Craig Robertson, Emanuele Trucco and Christopher K. I. Williams. 2014. *Dictionary of Computer Vision and Image Processing (2nd ed)*. John Wiley & Sons.

Kunihiko Fukushima. 1980. Neocognitron: a self-organizing neural network model for a mechanism of pattern recognition unaffected by a shift in position. *Biological Cybernetics* 36, 193–202.

Melvyn A. Goodale. 2011. Transforming vision into action. *Vision Research* 51, 1567–1587.

Simon Haykin. 2009. *Neural Networks and Learning Machines (3rd ed)*. Pearson Education.

ImageNet. http://image-net.org/about-overview. Checked 16.2.2016.

Nicola Jones. 2014. The learning machines. *Nature* 505, 146–148.

Martti Juhola. 2015. Neurolaskenta-kurssin materiaali. Informaatiotieteiden yksikkö, Tampereen yliopisto.

Andrej Karpathy. 2015. Convolutional Neural Networks for Visual Recognition. Online notes for the Stanford CS231n course. http://cs231n.github.io Checked 16.2.2016.

Alex Krizhevsky, Ilya Sutskever and Geoffrey Hinton. 2012. ImageNet classification with deep convolutional neural networks. In: *Advances in Neural Information Processing Systems* (NIPS 2012) 25, 1097–1105. MIT Press.

Norbert Krüger, Peter Janssen, Sinan Kalkan, Markus Lappe, Ales Leonardi, Justus Piater, Antonio J. Rodríguez-Sánchez, and Laurenz Wiskott. 2013. Deep hierarchies in the primate visual cortex: what can we learn for computer vision? *IEEE Trans. on Pattern Analysis and Machine Intelligence* 35, 8, 1847–1871.

Yann LeCun, Yoshua Bengio and Geoffrey Hinton. 2015. Deep learning. *Nature* 521, 436–444.

Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. 1998. Gradient-based learning applied to document recognition. In: *Proc. IEEE* 86, 11. 2278–2324.

Y. LeCun, L. Jackel, B. Boser and J. Dencker. 1989. Handwritten digit recognition: Applications of neural network chips and automatic learning. *IEEE Communications Magazine* 27, 11, 41–46.

Min Lin, Qiang Chen and Shuicheng Yan. 2014. Network in network. arXiv: 1312.4400v3.

Aravindh Mahendran and Andrea Vedaldi. 2014. Understanding deep image representations by inverting them. arXiv: 1412.0035v1.

Anh Nguyen, Jason Yosinski and Jeff Clune. 2014. Deep neural networks are easily fooled: high confidence predictions for unrecognizable images. arXiv: 1412.1897v4.

Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg and Li Fei-Fei. 2015. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision* 115, 3, 211–252.

Stuart Russell and Peter Norvig. 2010. *Artificial Intelligence – A Modern Approach (3rd ed)*. Pearson Education.

Jürgen Schmidhuber. 2015. Deep learning in neural networks: An overview. *Neural Networks* 61, 85–117.

Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke and Andrew Rabinovich. 2014. Going deeper with convolutions. arXiv: 1409.4842v1.

John K. Tsotsos. 1990. Analyzing vision at the complexity level. *Behavioral and Brain Science* 13, 423–469.

Vuorovaikutuksen osuudesta tiedon visualisoinnissa

Jukka Salonen

Tiivistelmä

Ihmisen on helpompi ymmärtää eli muodostaa näkemys kuvallisessa muodossa olevasta tiedosta esimerkiksi pelkkään numeeriseen tietoon verrattuna. Kun visuaalinen tieto tuotetaan ihmiselle teknologian avulla, on ihmisen ja järjestelmän välille mahdollista rakentaa vuorovaikutteinen käyttöliittymä. Tämä tutkielma selvittää vuorovaikutuksen osuutta tiedon visualisoinnissa. Työssä esitellään yleisimpiä kirjallisuudessa kuvattuja vuorovaikutusmenetelmiä, joita käytetään abstraktia tietoa visualisoitaessa. Erilaisia menetelmiä käyttäen voidaan tietojoukosta etsiä ja korostaa kohteita sekä yhteyksiä, jotka muutoin jäisivät havaitsematta.

Avainsanat ja -sanonnat: interaktio, taksonomia, käytettävyys, käyttöliittymä.

1. Johdanto

Teknologian avulla tuotetun tiedon visualisointi voidaan jakaa karkeasti tiedon esittämiseen (*representation*) ja käyttäjän vuorovaikutukseen (*interaction*) käsiteltävän tiedon tai järjestelmän kanssa [Yi *et al.* 2007]. Vuorovaikutus liittyy tässä ihmisen ja teknologian välisiin tapahtumiin teknologian tarjoaman käyttöliittymän välityksellä. Vuorovaikutuksen tutkimus on suhteellisen nuorta, ja käsitteistön luokittelu sekä metodologia on vielä melko vakiintumatonta [Kosara *et al.* 2003; Yi *et al.* 2007]. Merkittävä osa tiedon visualisointitutkimuksesta on keskittynyt erilaisten visualisointitapojen käsittelyyn, kun taas vuorovaikutuksen osuus jää vähäisemmälle huomiolle.

Eräs varhaisimpia vuorovaikutteisia järjestelmiä oli Fowlkesin todennäköisyyskaavio vuodelta 1969, jossa käyttäjä pystyi vaikuttamaan reaaliajassa kuvaajan sisältöön parametria muuttamalla. Vasta 1980-luvun puolivälissä edullisempien ja tehokkaampien työasemien ja näyttöjen tulo markkinoille vauhditti uusien ohjelmistojen (kuten Xgobi ja MacSpin) kehitystä. [Friendly 2006]

Tutkimusmenetelmänä on käytetty kirjallisuuskatsausta. Tutkielman alussa (luku 2) tarkastellaan, mitä on vuorovaikutus ja miten tietotyypit jaotellaan sekä ja esitellään yleisimpiä vuorovaikutustekniikoiden luokitteluja käytettävyyttä tutkivassa kirjallisuudessa. Tämän jälkeen (luku 3) perehdytään yksityiskohtaisemmin luokittelussa mainittuihin vuorovaikutustekniikoihin. Luvussa 4

on pohdintaa kirjoitelman tekniikoista ja luokituksista. Lopuksi luvussa 5 esitetään joitakin kehityssuuntia ja sovellusmahdollisuuksia.

2. Vuorovaikutuksen peruskäsitteitä ja luokituksia

Interaktiivisuus liittyy informaation visualisointiin. Informaation visualisointi on monimutkaisesta tiedosta tietokoneella tuotettua grafiikkaa, joka on tyypillisesti vuorovaikutteista ja dynaamista [Preece *et al.* 2015, 182]. Tavoitteena on Cardin ja kumppaneiden [1999] mukaan parantaa tiedon visualisoinnilla ihmisen kognitiota, mahdollistaa käyttäjä näkemään toistuvia malleja, kehityssuuntia ja poikkeamia sekä tämän seurauksena lisätä näkemystä esitettävästä tiedosta.

Spence [2007] ei halua visualisoinnin rajoittuvan pelkästään tietokoneella tuotettuun materiaaliin. Hän esittää visualisoinnin pelkästään ihmisen kognitiiviseksi toiminnaksi, jonka lopputuloksena on tiedosta luotu mentaalimalli. Joten näkemys ja mentaalimalli voidaan yhtä hyvin muodostaa staattisestakin kuvasta. [Spence 2007, 5]

Lyhimmillään vuorovaikutuksen määritelmän voi pelkistää "*kommunikoinniksi käyttäjän ja järjestelmän välillä*" [Dix *et al.* 2004, 124]. Shneiderman [1996] laajentaa vuorovaikutuksen roolia tunnetussa visuaalisen tiedonhaun mantrassaan: "*Ensin yleiskatsaus, tarkenna ja suodata, sitten yksityiskohdat tarpeen mukaan*". Aluksi luodaan käsitys siitä, mistä tietoa lähdetään etsimään. Sen jälkeen on tärkeää valita usein isostakin tietomassasta oleellinen käyttämällä tarkentamista ja suodatusta (katso kohta 2.2). Kukin näistä vaiheista vaatii vuorovaikutusta.

Käytettävyystutkimuksessa käytetään yleisesti Normanin seitsemänvaiheista vuorovaikutusmallia [Norman 2013, 41]. Käyttäjä laatii ensin toimenpidesuunnitelman, joka sitten suoritetaan laitteen käyttöliittymässä. Toimenpiteiden suorittamisen jälkeen käyttäjä havainnoi suunnitelman onnistumista ja tekee jatkosuunnitelman. Malli jakautuu tavoitteeseen, kolmeen toimeenpanovaiheeseen sekä kolmeen arviointivaiheeseen seuraavasti:

1. Muodostetaan tavoite.
2. Suunnitellaan toimenpide.
3. Määritellään toimenpiteiden sarja.
4. Suoritetaan toimenpiteiden sarja.
5. Havainnoidaan järjestelmän tila.
6. Tulkitaan havainto.
7. Verrataan lopputulosta tavoitteeseen.

Waren [2012] mukaan vuorovaikutus voidaan nähdä episteemisinä toimintoina: ”*episteeminen toiminto on toimintaa, jonka tarkoituksena on paljastaa uutta tietoa*”. Hyvä visualisointi sallii käyttäjän tarkentaa ja löytää lisää tietoa mistä tahansa, joka näyttää kiinnostavalta. Edelleen Ware näkee vuorovaikutteisen visualisoinnin kolmena toisiinsa lomittuneena palautesilmukkana, jossa kutakin silmukkaa vastaa oma prosessi. Ensimmäinen, matalimman tason silmukka, sisältää kohteen valinnan ja käsittelyn. Keskitason silmukka koostuu tietoavaruuden tutkimisesta ja navigoinnista. Korkeimmalla tasolla sijaitsee ongelmanratkaisusilmukka, jossa tiedosta muodostetaan hypoteesi. [Ware 2012, Ch. 10]

Vuorovaikutus voidaan jakaa toiminnan jatkuvuuden perustella kahteen tilaan: erilliseen (discrete) ja jatkuvaan (continuous). Erillistä vuorovaikutusta on esimerkiksi www-sivun linkin avaus. Vuorovaikutus on jatkuvaa, jos käyttäjän toiminta saa järjestelmän tuottamaan jatkuvaa palautetta. Jatkuvasta vuorovaikutuksesta on esimerkkinä mitta-asteikon muuttaminen reaaliaikaista tietoa tuottavaan viivakuvaajaan. [Spence 2007]

Seuraavaksi tarkastellaan joitakin tietotyyppien ja vuorovaikutustekniikoiden luokituksia. Luokituksia käyttämällä on pyritty hahmottamaan suunnitteluavaruutta, jossa vuorovaikutus tapahtuu.

2.1 Tietotyyppien luokittelu

Visualisoinnilla kuvattava tieto voidaan ryhmitellä usealla eri perusteella. Tiedon ulottuvuus voi tarkoittaa visualisoinnissa esimerkiksi avaruudellista ulottuvuutta (1D, 2D, 3D tai 4D eli 3D + aika) tai visualisoitavan tietorakenteen ulottuvuutta [Card *et al.* 1999]. Shneiderman [1996] määrittää kuvattavan tiedon sen ulottuvuuden mukaan seuraavasti:

- **Yksiulotteinen:** lineaariset ja peräkkäisesti esitettävät tietotyypit, kuten aakkosjärjestyksessä oleva nimilista tai ohjelman lähdekoodi.
- **Kaksiulotteinen:** tasoa kuvaava tieto, karttatieto, huoneiston pohjakartta tai muu pohjapiirustus.
- **Kolmiulotteinen:** fyysiset kohteet, kuten molekyylit, ihmiskeho tai rakennus.
- **Ajallinen:** aikajanat, projektisuunnitelmat tai historialliset esitykset.
- **Moniulotteinen:** relationaaliset ja tilastolliset tietokannat.
- **Puu:** hierarkiat tai puurakenteet, joissa juurisolmua lukuun ottamatta kukin solmu sisältää yksilöivän esivanhempisolmun.
- **Verkko:** graafi, tietoverkko, World Wide Web.

Spence [2007] jaottelee esitettävän tiedon kahteen kategoriaan: arvot *(values*) ja suhteet (*relations*). Arvot perustuvat tilastollisten muuttujien tyyppeihin. Spencen arvotyyppisen tiedon luokat ovat yhden (*univariate*), kahden (*bivariate*), kolmen (*trivariate*) sekä monen muuttujan (*multivariate*) tieto. Suhdetyyppisen tiedon Spence [2007] jakaa viivoihin, karttoihin ja diagrammeihin sekä puuesitykseen.

2.2 Klassinen tehtäväluokittelu

Klassinen vuorovaikutustekniikoiden luokittelu perustuu tietotyyppiperustaiseen tehtävänjakoon (TTT eli *type-by-task-taxonomy*) [Shneiderman 1996]. Shneidermanin [1996] tiedonhaun mantraa käsiteltiin jo edellä. Hän jakaa mantrassaankin lueteltuja tehtäviä seitsemään luokkaan:

- **Yleiskatsaus:** luodaan näkymä koko tietojoukkoon.
- **Tarkennus:** tarkennetaan näkymä kiinnostaviin kohteisiin.
- **Suodatus:** poistetaan joukosta ei-kiinnostavat kohteet.
- **Yksityiskohdat tarvittaessa:** valitaan kohteen tai ryhmän tiedot tarvittaessa.
- **Yhdistäminen:** tarkastellaan kohteiden välisiä yhteyksiä.
- **Historia:** ylläpidetään toimintojen historiatietoja, jotta mahdollistetaan toimintojen peruutus, toisto ja kohteen muu muokkaus.
- **Poiminta:** mahdollistetaan poiminta tiedon osajoukoista ja kyselyparametreista.

2.3 Tarkoitusperustainen luokittelu

Klassista vuorovaikutustekniikoiden luokittelua on pyritty kehittämään kattavampaan ja järjestelmällisempään suuntaan. Yin ja kumppanien [2007] menetelmänä oli kerätä laaja lista vuorovaikutustekniikoita kaupallisista ohjelmista ja julkaistuista artikkeleista. Aineistona oli 59 tutkimusartikkelia, 51 järjestelmää ja 311 yksittäistä vuorovaikutustekniikkaa. Seuraavana vaiheena he yhdistivät ja ryhmittelivät samankaltaisia tekniikoita omiin kategorioihin. Jakoperustaksi asetettiin se, mitä käyttäjä voi saavuttaa tietyllä vuorovaikutustekniikalla, eikä niinkään tekniikan toimintaperiaate. Esimerkiksi ympyräkaavio ja puukartta voivat näyttää erilaisilta, mutta silti ne palvelevat samaa käyttötarkoitusta: useampien yksityiskohtien saamista esityksestä. Lopputuloksena oli seitsemän tarkoituslähtöistä luokkaa. [Yi *et al.* 2007]

Ensimmäinen Yin ja muiden luokista on *valitseminen*. Jokin asia merkitään kiinnostavaksi, jotta sitä olisi helpompi myöhemmin seurata. Valinnan käyttäminen edellyttää, että seurattavat kohteet erottuvat muusta tietojoukosta esimerkiksi värityksellä, koolla, muodolla tai kirjanmerkillä.

Toinen luokka on *tutkiminen*. Katsotaan jotain muuta tietojoukkoa, koska näytön rajalliselle pinta-alalle mahtuu vain osa tiedosta. Esimerkkinä on karttapohjan näytön vieritys joko hiirellä tai vierityspalkin avulla [Yi *et al.* 2007].

Seuraava luokka on *uudelleenkonfigurointi*, jossa näytetään tieto uudessa järjestyksessä. Uudelleenkonfiguroinnin sovellusesimerkkeinä on pistekaavion XY-akseleissa näytettäviä muuttujien vaihtaminen tai histogrammissa näytettävien palkkien järjestyksen muuttaminen [Yi *et al.* 2007].

Koodaa-luokassa käyttäjä voi muuttaa näytettävän tiedon peruselementtejä kuten kohteiden ulkoasua. Valitulla ulkoasulla, kuten väreillä ja muodoilla, saadaan esityksestä tuotua esiin ymmärtämistä helpottavia asioita. Esimerkiksi maanpinnan korkeuserot voidaan kuvata värispektriä käyttäen, jolloin maanpinnan muodot ovat helpompia hahmottaa kognitiivisesti [Yi *et al.* 2007].

Viidennessä *tiivistäminen/tarkennus*-luokassa näytetään tiedosta joko enemmän tai vähemmän yksityiskohtia. Näin voidaan antaa tiedosta yleiskuva tai tutkia yksityiskohtia. Esimerkki *tiivistäminen/tarkennus* -tekniikasta on kartan tarkennus tai loitonnus. Tieto säilyy jatkuvasti samana, vain esityksen skaalaus muuttuu. Toinen tekniikkaesimerkki on tooltip-ikkunan tulostus valitusta kohteesta siirrettäessä hiiren kursori kohteen päälle. Erillisessä ikkunassa voidaan näyttää yksityiskohtaista tekstimuotoista tietoa, joka ei muuten mahtuisi esitykseen [Yi *et al.* 2007].

Suodatus-luokassa näytetään tietoa ehdollisesti. Käyttäjä valitsee näytettävät kohteet rajaamalla sallitut arvot tietylle välille tai asettamalla ehtoja. Asetus voi tapahtua esimerkiksi liukusäätimellä tai valintalaatikolla. Tekniikka mahdollistaa dynaamisen kyselyn käytön: säätimen muutos vaikuttaa välittömästi esityksen sisältöön [Yi *et al.* 2007].

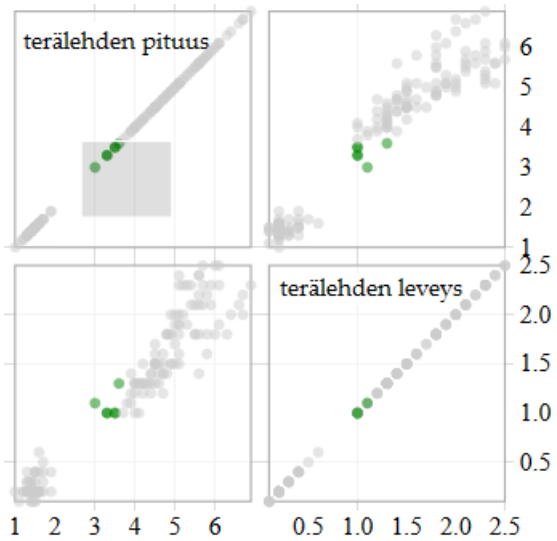
Viimeisessä *yhdistäminen*-luokassa näytetään yhteen kuuluvat kohteet korostamalla tietojen välistä suhdetta tai yhteyttä. Yhdistämistekniikkaa voidaan soveltaa myös näyttämään piilossa olevia tietoja, joilla on suhde tarkasteltavaan kohteeseen. Esimerkkisovelluksena on samasta tietojoukosta tehdyn kahden erityyppisen esityksen linkittäminen. Esityksessä valittu tieto näytetään korostettuna myös toisentyyppisessä esityksessä, johon linkitys on tehty [Yi *et al.* 2007].

3. Yleisimmät vuorovaikutustekniikat

Kuvattavien tekniikoiden valinta pohjautuu kohdassa 2.3 esitettyyn tehtävien tarkoitukseen perustuvaan luokitteluun [Yi *et al.* 2007]. Järjestys on keinotekoinen, koska usein samassa kuvaajassa käytetään useita menetelmiä samaan aikaan.

Valitse (Select)

Tietojoukosta valitaan haluttu määrä kiinnostavia alkioita. Usein valitut alkiot erotetaan muista alkioista korostamalla niiden ulkoasua eri värillä tai muodolla. Kuvassa 1 on valitut pisteet väritetty vihreällä.



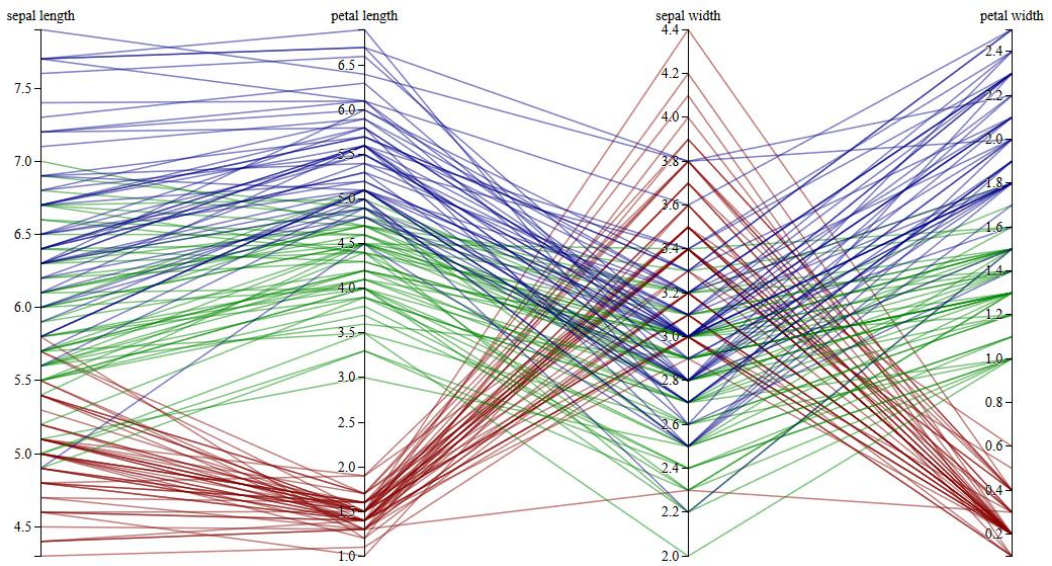
Kuva 1. Pisteiden valitseminen neljän näkymän kuvaajajoukossa. Valinta on tehty ylhäällä vasemmalla olevaan terälehden pituus -kuvaajaan. Valittujen pisteiden joukko näkyy korostettuna vihreällä värillä myös kolmessa muussa eri ominaisuutta esittävässä kuvaajassa [D3.js].

Tutki (Explore)

Vieritys (*panning*) on yleisimmin käytössä oleva tutkiva vuorovaikutustekniikka. Sen avulla voidaan rajallinen näyttötila hyödyntää paremmin. Tarkennuksella saadaan rajatusta alueesta suurennettua tarkempi kuva. Vierityksen avulla on mahdollista siirtää näytettävää kehystä jonkun suuremman alueen, kuten kartan, sisällä.

Uudelleenkonfiguroi (Reconfigure)

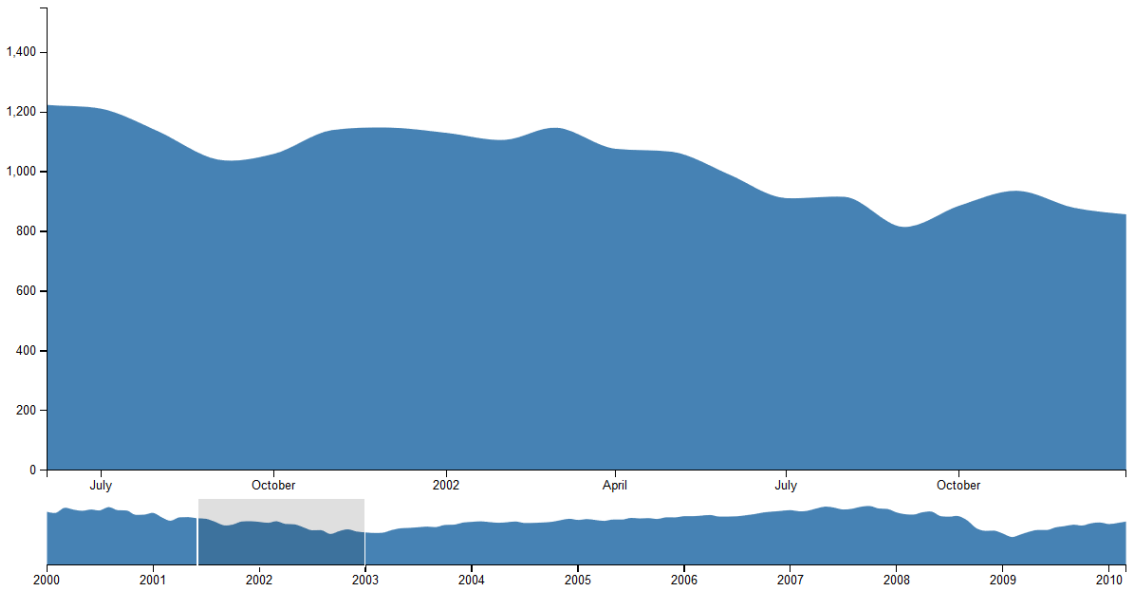
Pistekaavioista on kehitetty sovellutuksia, joiden avulla voidaan esittää kaksiulotteista tietoa useampia ulottuvuuksia. Ulottuvuuksien visualisointi tapahtuu lisäämällä samansuuntaisia koordinaatteja, kuten Y-akseleita, jokaista uutta ulottuvuutta kohti. Rinnakkaiset koordinaatit (Parallel Coordinates) [Inselberg and Dimsdale 1987] on kaaviotyyppi moniulotteisen ja -muuttujaisen tiedon esittämiseen. Se (kuva 2) mahdollistaa muun muassa harjauksen, vierityksen ja akseleiden uudelleenjärjestelyn.



Kuva 2. Rinnakkaiset koordinaatit. Kuvassa on havaintoja kolmen eri Iris-lajikkeen kukista [D3.js].

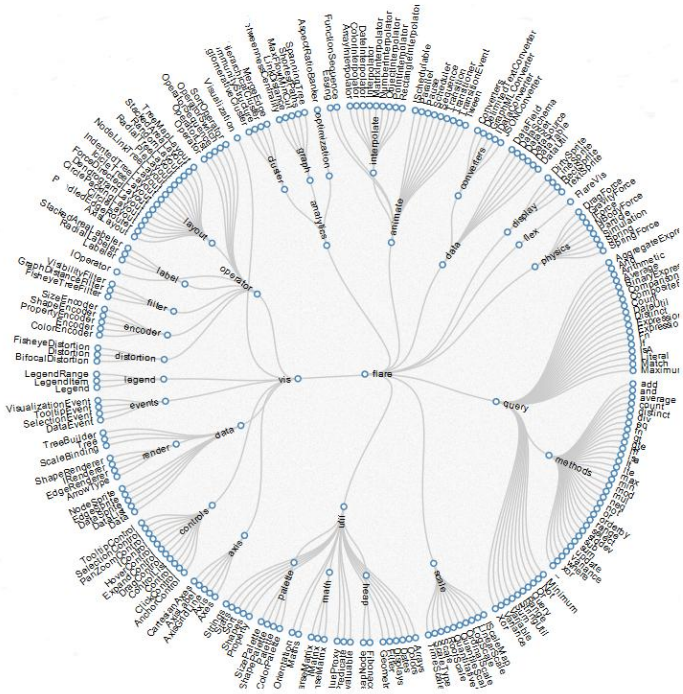
Suodata (Filter)

Suodattamalla valitaan näkymässä olevasta tietojoukosta uusi osajoukko asettamalla ehtoja. Menetelmällä saadaan vähennettyä näytettävien kohteiden lukumäärää ja olennaisen tiedon osuutta vastaavasti lisättyä. Yksi käytetyimmistä ratkaisuista on Focus+Context. Esitettävä tieto ei useinkaan mahdu kokonaan näytölle. Toisinaan halutaan tietoa vain osasta tietojoukkoa (focus) säilyttäen silti yhteys muuhun esitykseen eli kontekstiin. Kuva 3 esittää dynaamisesti valitun kohteen ja valinnan jälkeisen näkymän.



Kuva 3. Focus+Context. Alemmassa osassa on valintaikkuna, jossa on näkyvissä koko konteksti. Käyttäjän valitsema aikajakso (fokus) näkyy harmaana laatikkona. Ylemmässä kuvassa on valitun aikajakson sisältö suurennettuna [D3.js].

Erilaisilla Focus+Context -tekniikoiden variaatioilla voidaan muuta, ei-olennaista tietoa häivyttää esimerkiksi sumentamalla (*distortion*) tai korostaa olennaista suurennuslasin tapaisilla kontrolleilla. Kuvassa 4 esitettävä hyperbolinen puu [Munzner and Burchard 1995] on yksi tapa toteuttaa sumentaminen. Puun rakenne muutetaan keskelle valitun alkion mukaiseksi.



Kuva 4. Hyperbolinen puu [D3.js].

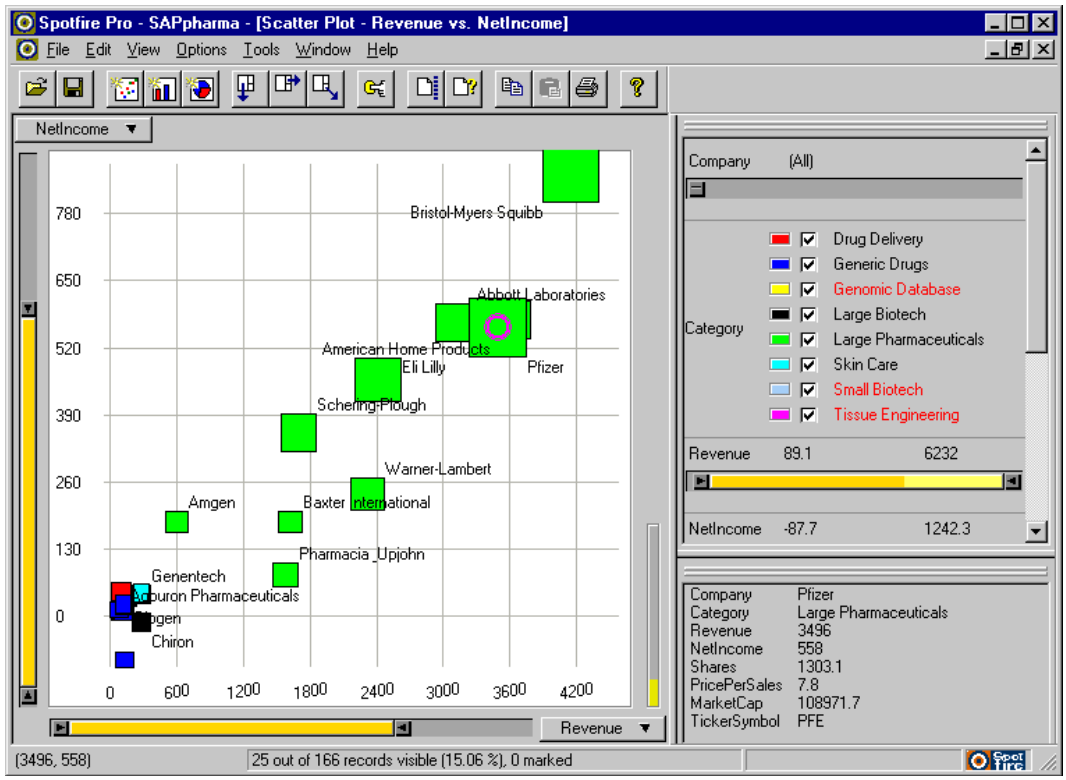
Käyttöliittymälle voidaan luoda samasta tiedosta useita näkymiä. Näkyminä voivat olla esimerkiksi histogrammi, toisessa näkymässä ympyräkaavio ja kolmannessa pistekaavio. Eri näkymät samasta tiedosta voidaan toteuttaa linkittämällä (*linking*) ne toisiinsa, jolloin yhteen näkymään tehty muutos vaikuttaa automaattisesti myös muihin näkymiin. Kuvassa 1 on esimerkki usean pistekaavion yhteenlinkittämisestä.

Kohteen valintaa ja tarkastelua kutsutaan harjaamiseksi (*brushing*) [Becker and Cleveland 1987]. Harjaaminen voidaan suorittaa suoraan esimerkiksi hiirellä, liukusäätimellä tai syöttökenttää käyttämällä. Lisäksi pistejoukolle voidaan suorittaa haluttuja toimintoja, kuten joukko-operaatioita, lisäyksiä tai poistoja.

Näytön tietoja voidaan muuttaa nopeasti käyttämällä näytöllä olevaa säädintä. Tällaista säädintä kutsutaan dynaamiseksi suodattimeksi (*dynamic filter*).

Dynaaminen kysely (*dynamic query*) [Ahlberg *et al*. 1992] on tehokas menetelmä tiedon esityksen dynaamiseen muuttamiseen (kuva 5), kun tieto haetaan

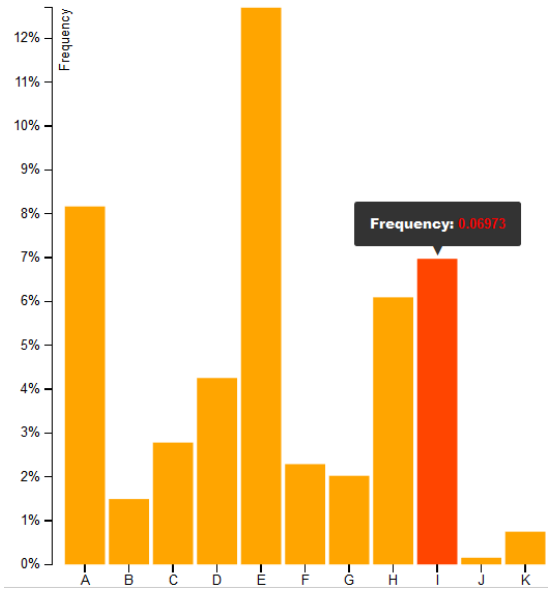
tietokantakyselyn kautta. Kysely käynnistyy usein suoraan esimerkiksi liukusäätimen tai valintaruudun avulla.



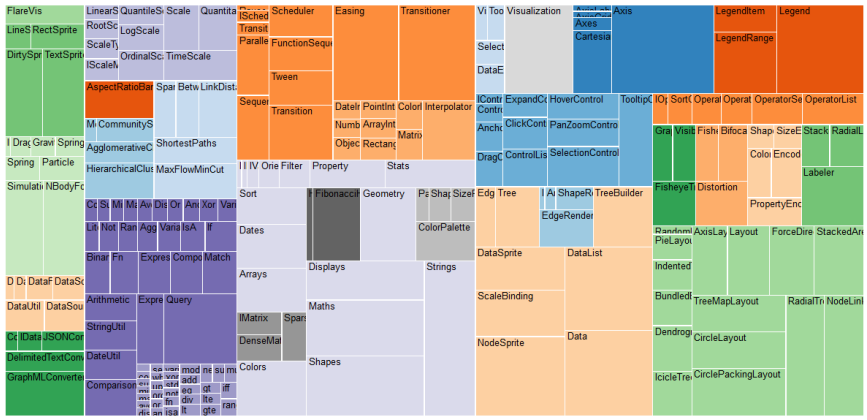
Kuva 5. Dynaaminen kysely Spotfire™-ohjelmalla [Spotfire 1999].

Tiivistäminen ja tarkentaminen (Abstract/Elaborate)

Tiivistämisellä ja tarkentamisella on tiedon esitystä muuttamalla mahdollista nähdä uusia yksityiskohtia poistaen näytöltä ylimääräisiä alkioita. Tiivistämisen vastakohtana on loitonnus, jonka avulla voidaan saada paremmin näkyviin yleisnäkymä tietomassassa. Kuvassa 6 käyttäjä on valinnut palkin, jonka seurauksena palkin yläosaan tulostuu lisätietoikkuna. Kuva 7 esittää Treemap-kuvaajaa, jossa ali-ikkunan valinnalla avautuu uusi näkymä. Tarkentamista ja loitonnusta käytetään myös karttatiedon tutkimisessa esimerkiksi Google Maps-sovelluksessa.



Kuva 6: Tooltip-ikkuna [D3.js].



Kuva 7: Treemap-kuvaaja [D3.js].

4. Keskustelu ja johtopäätökset

Kirjallisuuskatsauksen perusteella suunnitteluavaruus näyttäytyy haastavana. Menetelmistä ja tekniikoista löytyi niukasti vakiintuneita interaktioiden suunnittelumalleja, joten suunnittelijan on haettava tarvittava tieto useasta eri lähteestä. Waren [2012] teos tarjoaa perustan fysiologisten periaatteiden ymmärtämiseen. Preecen [2015] kirja keskittyy yleisesti hyvän vuorovaikutteisuuden aikaansaamiseen. Spence [2007] lähestyy aihetta myös monipuolisesti ja on yksi perusteoksista interaktiota suunniteltaessa. Lisäksi pätevät edelleen hyvän käytettävyyden lainalaisuudet [Norman 2013].

Kohdassa 2 esiteltiin Shneidermanin [1996] visuaalisen tiedonhaun mantra. Myöhemmin Shneiderman [Card *et al.* 1999] tarkensi kuuluisan mantra-artikkelin olevan enemmän kuvaileva ja havainnollistava kuin ohjaileva. Craft ja Cairns [2005] tutkivat vuoden 1996 artikkelin käyttöä muiden julkaisemissa

tieteellisissä artikkeleissa. He luokittelivat artikkelien viittaukset viiteen eri kategoriaan: toteutukset, metodologiat, evaluoinnit, luokittelut ja muut.

Toteutukseen liittyvissä artikkeleissa huomattiin, että Shneidermanin mantraa tai tietotyyppipohjaista (TTT eli *type-by-task-taxonomy*) luokittelua käytettiin parempien metodologisten oppaiden puutteessa. Lisäksi todettiin että mantran selkeältä kuulostavat suositukset kannustivat sen käyttöön. [Craft and Cairns 2005]

Metodologiaan liittyvien artikkelien kirjoittajien mielestä Shneidermanin karkea rakenne salli heidän luoda tarkemman ja tutkimuskohteesta riippuvamman mallin [Craft and Cairns 2005].

Evaluoinneista kirjoittaneista tutkijoista osa koki Shneidermanin käyttämien tietotyyppien tunnistamisen olevan tärkeää, koska ne tekevät samanlaisuuksien ja eroavaisuuksien vertailun helpommaksi eri visualisointityyppien välillä [Craft and Cairns 2005].

Luokitteluaiheen valinneista artikkelinkirjoittajista kokivat erottelun tieteellisen ja abstraktin tiedon visualisoinnin välillä turhaksi [Craft and Cairns 2005].

Craft ja Cairns [2005] suosittelivat mantraa lopulta lähinnä joihinkin yksinketaisiin suunnitteluongelmiin. He korostivat myös empiiristen lisätutkimusten tarvetta. Samalla he esittivät tarpeen kokonaisvaltaiselle (holistic) suunnittelumetodologialle sekä hajanaisten ja erillisten oppaiden korvaamista tutkituilla suunnittelumalleilla.

Keim ja muut [2008] muotoilivat Shneidermanin mantraa sopivammaksi visuaalisen analytiikan prosessiin. Koska visuaalisessa analytiikassa käsiteltävä tieto on monimutkaista ja määrällisesti suurta, sille suoritetaan käsittelyä sekä ennen vuorovaikutteista esitystä että myös sen jälkeen. Tiedosta ei voida luoda aluksi yleiskatsausta, vaan sen määrää ja monimutkaisuutta on ensin laskettava ihmisen vastaanottokyvylle sopiviin rajoihin. Visuaalisen analytiikan mantra kuuluukin: ”*Analysoi ensin – näytä tärkein – tarkenna, suodata ja analysoi lisää – yksityiskohdat tarpeen mukaan*”. [Keim *et al*. 2008]

5. Yhteenveto

Tutkielmassa esiteltiin vuorovaikutuksen määritelmiä ja yleisimpiä vuorovaikutustapojen ja visuaalisen tiedon luokitteluja. Myös yleisimpiä vuorovaikutustekniikoita tarkasteltiin tiedon visualisoinnin kannalta. Keskusteluosuudessa pohdittiin Shneidermanin visuaalisen tiedonhaun mantraan kohdistunutta polemiikkia.

Tiedon määrän jatkuvasti lisääntyessä myös sen visualisoinnille tulee kysyntää. Erilaisia tiedon visualisointitapoja on lukematon määrä ja niiden edel-

leenkehitys on nopeaa. Vuorovaikutteisten sovellusten ja infograafisen tiedon tuottajille on olemassa sekä valmiita työkaluohjelmia että kehityskirjastoja [D3.js; Processing]. Myös uusien syötteenantomenetelmien eli modaliteettien käyttöönotto on ajankohtaista. Uudet päätelaitteet ovat pienikokoisia eikä perinteisen näppäimistön ja hiiren käyttö ole useinkaan mahdollista. Puheen soveltamista (Siri-ohjelma) on jo kokeiltu menestyksekkäästi mobiililaitteilla.

Tiedon louhinta ja siihen liittyvä analytiikka sekä koneiden ja asioiden Internet (IoT) ovat kiinnostavia uusia tutkimuskohteita. Perinteisesti suljettuina pidetyt tietovarastot, kuten kartta-, liikenne- ja päätöksentekotiedot, ovat pian jokaisen ulottuvilla. Tämä avaa uusia mahdollisuuksia tiedon hyväksikäyttöön ja ymmärtämiseen myös visualisoinnin avulla.

Viiteluettelo

Ahlberg, C., Williamson, C. and Shneiderman B. 1992. Dynamic Queries for Information Exploration: An Implementation and Evaluation, In: *Proc. of the SIGCHI Conference on Human Factors in Computing Systems,* 619—626.

Becker, R.A and Cleveland, W.S. 1987. Brushing scatterplots. *Technometrics,* 29.2, 127—142.

Card, S.K, Mackinlay J.D. and Shneiderman, B. 1999. *Readings in Information Visualization: Using Vision to Think*. Morgan Kaufmann.

D3.js. 2016. Data-Driven Documents JavaScript library. https://github.com/mbostock/d3/wiki/Gallery Checked 24.2.2016.

Dix, A.J., Finlay, J., Abowd, G. D. and Beale, R. 2004. *Human-Computer Interaction*. Pearson Prentice Hall.

Craft, B. and Cairns, P. 2005. Beyond Guidelines: What Can We Learn from the Visual Information Seeking Mantra? In: *Proceedings of the Ninth International Conference on Information Visualisation*, 110-118.

Friendly, M. 2006. A Brief History of Data Visualization. In: *Handbook of Computational Statistics: Data Visualization*. Springer.

Inselberg, A. and Dimsdale, B. 1987. Parallel coordinates for visualizing multi-dimensional geometry. In: *Proc. of Computer Graphics International '87*, 25—44. Springer.

Keim, Daniel A., Mansmann, F., Schneidewind, J., Thomas, J. and Ziegler, H. 2008. *Visual Data Mining: Theory, Techniques and Tools for Visual Analytics*. Springer.

Kosara, R., Hauser, H. and Gresh, D. L. 2003. An interaction view on information visualization. In: *Proceedings of EUROGRAPHICS 2003*, 123—137.

Munzner, T. and Burchard, P. 1995. Visualizing the structure of the world wide web in 3d hyperbolic space. In: *Proceedings of the 1995 Symposium on Virtual Reality Modeling Language (VRML'95)*, 33—38. ACM Press.

Norman, D. 2013. *Design of Everyday Things : Revised and Expanded Edition*. Basic Books.

Processing. 2016. Processing.org. https://processing.org/ Checked 24.2. 2016.

Preece, J., Sharp H. and Rogers Y. 2015. *Interaction Design: Beyond Human-Computer Interaction*. Wiley.

Shneiderman, B. 1996. The eyes have it: A task by data type taxonomy for information visualization. In: *Proceedings Visual Languages.* IEEE Computer Society Press, 336—343.

Spence, R. 2007. *Information Visualization: Design for Interaction*. Pearson Prentice Hall.

Spotfire. 1999. Dynamic queries, starfield displays, and the path to Spotfire. http://www.cs.umd.edu/hcil/spotfire/ Checked 24.2.2016.

Ware, C. 2012. *Information Visualization: Perception for Design*. Morgan Kaufmann.

Yi, J. S., ah Kang, Y., Stasko, J. and Jacko, J. 2007. Toward a deeper understanding of the role of interaction in information visualization. *IEEE Transactions on Visualization and Computer Graphics* 13(6), 1224—1231.

Katsaus strategiapelien vastustajien tekoälyn kehityspotentiaaliin

Tuomo Sillanpää

Tiivistelmä.
Yksi tekoälyohjelmoinnin yleisesti tunnetuimmista sovellusalueista, etenkin nuorempien sukupolvien keskuudessa, on elektronisissa peleissä. Yksinkertaisissakin peleissä on usein toteutettuna pelaajan vastustajien toimintaa ohjaava logiikka, joka voidaan luokitella alkeelliseksi tekoälyksi. Monimutkaisemmissa ja moniulotteisemmissa peleissä, kuten vuoropohjaisissa ja reaaliaikaisissa stategiapeleissä, tekoälyn ohjaaman vastustajan tai vastustajien toiminta nousee keskeiseksi osaksi pelin haastavuutta ja mielekkyyttä. Verrattain hyvin toteutettunakaan perinteinen tekoälyvastustaja ei usein kykene tarjoamaan varteenotettavaa haastetta noviisitasoa kehittyneemmille pelaajille. Tästä syystä onkin tarvetta kehittää metodeja, joilla tekoäly kykenee tarjoamaan mielekkään haasteen myös eksperttitason pelaajille.

Avainsanat ja -sanonnat: koneoppiminen, tekoäly, tekoälyohjelmointi, strategiapeli

1. Johdanto

Tässä tutkielmassa tarkastellaan kehityspotentiaalia monimutkaisten elektronisten pelien vastustajien tekoälyn kyvykkyyden kehittämisessä. Vastustajilla tarkoitetaan tässä kontekstissa pelin sellaisia elementtejä, jotka pyrkivät estämään pelaajaa saavuttamaan tavoitettaan pelin sisällä, eli voittamaan pelin. Yksinkertaisen tasohyppelypelin tapauksessa tällainen vastustava tekoäly saattaa esimerkiksi ohjata hirviöitä, jotka pyrkivät tuhoamaan tasolla etenevän pelaajan hahmon. Tällaisessa pelissä tekoälyn toiminta on kuitenkin useimmiten yksinkertaista ja ennalta-arvattavaa, sillä pelin mielekkyyden ei ole suunniteltu perustuvan tekoälyn haastavuuteen, vaan pikemminkin muihin peliteknisiin seikkoihin, kuten vastustajien määrälliseen ylivoimaan tai pelaajan ohjausliikkeiden oikeelliseen toteutukseen.

Erityispainotus tässä katsauksessa on strategiapelien tekoälyssä, sillä strategiapelien peliskenaariot mallintavat usein tilanteita, joissa pelaajan on huomioitava useita eri muuttujia sekä omansa että vastustajiensa tilanteen suhteen sekä kyettävä mukauttamaan omaa strategiaansa vastustajan toiminnan mu-

kaan ollakseen voitokas. Strategiapelit ovat myös sikäli mielekäs tekoälytutkimuksen kohde, että ihmispelaajat ja tekoälyn ohjaamat pelaajat ovat usein identtisessä asemassa pelin toiminnan suhteen. Tällöin pelin kannalta ainoa ero ihmispelaajan ja tekoälypelaajan välillä on toimintaa ohjaavassa logiikassa, ihmisaivot vastustajanaan peliin ohjelmallisesti toteutettu tekoäly.

Luvussa 2 käsitellään strategiapelien yleistä rakenetta, pelaajien tavoitteita ja metodeja niiden saavuttamiseen. Lisäksi kuvataan eräitä suosittuja strategiapelejä ja perinteisen tekoälyn toimintaa strategiapeleissä yleisesti sekä erikseen esiteltyjen pelien tapauksessa. Luvussa 3 tutustutaan erinäisiin tekoälyn kehitystekniikoihin erityisesti strategiapelien tekoälyn toiminnan kehittämisen näkökulmasta. Lopuksi pohditaan esitettyjen metodien realistista toteutettavuutta strategiapeleihin yleisesti ja tekoälyn kehittymisen mahdollista vaikutusta strategiapeligenren kehitykseen tulevaisuudessa.

2. Perinteinen tekoäly strategiapeleissä

Tässä luvussa kuvataan useimmat strategiapelimuodot yleisellä tasolla sekä tutustutaan hieman yksityiskohtaisemmin muutamaan suosittuun strategiapeliin. Lisäksi käsitellään perinteisen tekoälyn yleisiä toimintatapoja, vahvuuksia ja heikkouksia kyseisissä pelimuodoissa ja peleissä.

2.1. Strategiapelien rakenne

Tässä tutkielmassa keskitytään elektronisten strategiapelien ehkäpä yleisimpään ja tekoälytoteutuksen kannalta mielenkiintoisimpaan muotoon, eli peleihin, jotka mallintavat kahden tai useamman tahon välistä sotatilannetta, pohjautuen teemallisesti joko todelliseen maailmaan ja sen konflikteihin, tai usein fiktiivisen maailman vastaaviin. Tällaisissa strategiapeleissä pelaajan lopullisena tavoitteena on useimmiten kukistaa vastustajansa aseellisesti.

Yleisesti strategiapelien kolme olennaisinta pelaajan hallinnoimaa elementtiä ovat resurssit, tuotantokyky ja yksiköt. Resursseja, kuten kultaa tai öljyä keräämällä pelaaja kehittää tuotantokykyään, esimerkiksi rakentamalla tehtaita, joiden avulla hän tuottaa tai parantelee yksiköitä, kuten esimerkiksi panssarivaunuja. Näillä yksiköillä pelaaja kykenee taistelemaan toisten pelaajien yksiköitä vastaan tai heikentämään heidän resurssienhankinta- ja tuotantokykyään valtaamalla resursseja tuottavia alueita tai tuhoamalla tuotantorakennuksia. Tuotantokykyyn voidaan laskea myös useaan strategiapeliin kuuluva teknologian kehittäminen, joka yleensä mahdollistaa kehittyneempien yksiköiden tuottamisen tai parantaa olemassa olevien yksiköiden suorituskykyä jollakin tavalla.

Sotastrategiapelit voidaan jakaa karkeasti kahteen eri aligenreen, vuoropohjaisiin ja reaaliaikaisiin. Resurssi-tuotanto-yksikkö -kolminaisuus on kuitenkin kummassakin aligenressä hallitseva teema.

2.2. Perinteinen tekoäly strategiapeleissä ja sen haasteet

Strategiapelien tietokonevastustajien tekoälyn kyvykkyyden merkitys riippuu vahvasti pelattavasta pelimuodosta. Monissa strategiapeleissä pelin tuotannossa on keskitetty suuri osa pelin kehitysresursseista yksinpelattavaan kampanjamoodiin, jossa pelaaja etenee pelissä suorittamalla tehtäviä pelin tarinaa edistävässä ennalta määritellyssä järjestyksessä. Tällöin tekoälyn suorituskyvyn merkitys jää vähäisemmäksi, sillä pelaaja ja tekoälyn ohjaamat ihmispelaajaa vastustavat voimat harvoin aloittavat samasta asemasta. Usein kampanjamoodissa suoritettavat tehtävät ovat etukäteen luotuja skenaarioita, joissa tekoälyä on voitu erikseen räätälöidä suorittamaan tarinan kannalta merkityksellistä toimintaa (kuten esimerkiksi partioimaan tiettyjä alueita tai hyökkäämään pelaajan kimppuun tiettynä aikana). Tällöin tekoäly ei yritäkään toimia ihmispelaajan tavoin.

Useimmissa strategiapeleissä on myös mahdollisuus pelata yksittäisiä otteluita muita pelaajia vastaan. Tällöin peli lähtökohtaisesti alkaa tasa-arvoisesta tilanteesta, jossa kullakin pelaajalla on käytössään sama määrä resursseja, yksiköitä ja tuotantokykyä, ja pelattavan pelialueen muut elementit, kuten maaston muodot tai resurssiesiintymien sijainnit, eivät anna merkittävää etua kenellekään yksittäiselle pelaajalle. Tässä pelimuodossa ihmispelaajan voi lähes kaikissa strategiapeleissä korvata myös tekoälyn ohjaamalla toimijalla. Tekoälypelaajan haastavuus on usein myös säädettävissä halutulle tasolle. Kuten Gemine ja muut [2012] toteavat, on haastavuuden korotus useimmiten kuitenkin toteutettu erinäisillä pelitilanteen tasa-arvoisuutta rikkovilla tekoälyvastustajalle annettavilla resurssi- tai muilla bonuksilla, älyllisen suorituskyvyn pysyessä muuttumattomana alempiin haastavuustasoihin nähden. Sin ja muiden [2014] mukaan tämäntyyppisen "tekoälyn huijauksen" tiedetään vaikuttavan negatiivisesti pelaajien pelikokemukseen.

Chioun [2007] mukaan strategiapelien tekoälyn toteutuksen haastavuus johtuu osittain siitä, että toisin kuin shakin ja tammen kaltaisissa peleissä, tekoälylle ei voida antaa *täydellistä tietoa* ennen pelin alkua ja pelin aikana (kuten esim. shakissa laudan rakenne, pelinappulat ja niiden sijainti), jolloin sille ei voida toteuttaa algoritmia, joka laskisi jatkuvasti optimaalisen taktiikan ihmispelaajaa vastaan. Tällöin tekoälyllä tulee olla hallussaan *strategista tietämystä* (strategic knowlegde). Tällaisen tietämyksen toteuttaminen on kuitenkin erittäin monimutkaista ja haastavaa.

2.2.1. Reaaliaikaiset strategiapelit

Reaaliaikaiset strategiapelit (RTS, Real-Time Strategy) poikkeavat vuoropohjaisista strategiapeleistä aika-rajoitteisuudestaan ja dynaamisuudestaan aiheutuvalla realismilla. RTS-pelien reaaliaikaisuus ja pelaajien toimintojen samanaikaisuus mahdollistavat myös yleisesti suuremman osallistujamäärän kuin mikä vuoropohjaisissa peleissä olisi mielekästä. Tästä syystä myös tarve älykkäille tekoälytoimijoille, sekä vastustajina että liittolaisina, on RTS-peleissä huomattava. [Chen *et al*. 2014]

Eräs ehkä tunnetuimmista ja suosituimmista RTS-peleistä on Blizzard Entertainment -peliyhtiön kehittämä tällä hetkellä kahdesta pelistä ja niiden lisäosista koostuva Starcraft-pelisarja. Starcraft-pelien ympärille on kehittynyt valtava määrä ammattimaista pelitoimintaa, ja suuri osa RTS-tekoälytutkimuksesta liittyykin joko suorasti tai epäsuorasti Starcraft-peleihin.

Starcraft-pelien ottelumuoto noudattaa suoraan kohdassa 2.1 kuvailtua yksiköt-tuotanto-resurssit -kaavaa. Kukin pelaaja pelaa jollakin pelin kolmesta eri puolueesta (rodusta): Terranit, Zergit ja Protossit. Kullakin rodulla on täysin omat yksikkönsä ja rakennuksensa, mutta pelit on pyritty tasapainottamaan niin, että kaikki rodut kykenevät korkeallakin taitotasolla tasavertaiseen kamppailuun. Rotujen ja niiden yksiköiden suuret keskinäiset erot tuovat kuitenkin lisää syvyyttä pelien strategisiin ja taktisiin kuvioihin ja lisähaasteen tekoälyn toteutukselle. Parkin ja muiden [2012] mukaan Starcraftiin on edelleenkin haastavaa laatia tekoälyä siinä esiintyvän suuren yksikkö- ja rakennusmäärän, resurssienhallinnan ja korkeatasoisten taktiikkavaatimusten vuoksi.

Synnaeve ja muut [2015] puhuvat RTS-pelien tekoälyn kolmesta eri ydinkomponentista: mikrohallinnasta, taktiikasta ja strategiasta. Tämä jako on yleisesti käytössä myös suosituimpien RTS-pelien (kuten Starcraft-sarjan pelien) pelaajien keskuudessa pelin ja pelaajien taitojen ja suoritusten eri aspekteista keskusteltaessa, joskin vaihtelevalla terminologialla. Näin ollen se soveltuu hyvin myös tekoälyn ominaisuuksien luokitteluun ja arviointiin. Mikrohallinnalla (micro-management) tarkoitetaan RTS-pelien kontekstissa yleisesti pelaajan kykyä hallita yksiköitään yksilö- tai pienryhmätasolla, useimmiten taistelutilanteessa. Mikrohallinnan merkitys korostuu esimerkiksi yhteenotossa, jossa pelaajan 1 käytössä olevilla yksiköillä on pidempi tulikantama kuin pelaajan 2 joukoilla. Tällöin on pelaajan 1 edun mukaista liikuttaa ja pysäyttää joukkojaan toistuvasti siten, että ne kykenevät tulittamaan pelaajan 2 joukkoja näiden voimatta vastata tuleen. Tällainen mikrohallinta vaatii ihmispelaajalta tilannekohtaisen ymmärryksen lisäksi myös nopeaa ja tarkkaa käsi-silmä-koordinaatiota

pelin hallintainstrumenttien (RTS-peleissä lähes aina näppäimistö ja hiiri) käytössä.

Taktiikalla viitataan RTS-peleissä useimmiten pelaajan armeijoiden hallintaan ja sijaintiin. Taktiikassa on siis kyse ylemmän luokan hallinnasta mikrohallintaan verrattuna. Vastustajan armeijan pakottaminen taisteluun tälle epäedullisessa maastossa on yksi esimerkki onnistuneesta taktisesta siirrosta.

Strategialla tarkoitetaan yleisesti niitä pelaajan ylemmän toiminnan elementtejä, jotka eivät kuulu taktiikan tai mikrohallinnan piiriin, kuten talouden ja teknologian kehitystä, tuotantokykyä ja tuotettujen armeijoiden koostumusta. Strategia onkin ehkä useimmiten juuri se osa-alue pelisuoritusta, jossa muilta osin verrattain päteväkin tekoäly jää auttamatta eksperttitason ihmispelaajan jalkoihin. Chioun [2007] mukaan yksi haastavimmista osa-alueista pelien tekoälyn kehittämisessä on ihmismäiseen strategiseen intuitioon kykenevän tekoälyn laatiminen.

2.2.2. Vuoropohjaiset strategiapelit

Aiemmissa alakohdissa kuvaillut pelaajan (ja myös tekoälyn) toiminnan komponentit pätevät useimmiten sellaisenaan myös moniin vuoropohjaisiin strategiapeleihin (TBS, Turn-Based Strategy), poislukien mikrohallinan käsite, jolle sellaisenaan harvoin löytyy vastinetta TBS-peleistä reaaliaikaisuudesta seuraavien aikarajoitteiden poissaolosta johtuen.

TBS-peleihin lukeutuu mm. suosittu Civilization-pelisarja, jossa pelaaja kehittää omaa sivilisaatiotaan historian alkuhämäristä nykypäivään asti, sekä Heroes of Might & Magic -sarjan pelit, jotka noudattavat melko läheisesti useista RTS-peleistä tuttua formaattia yksikkötuotannosta ja vastustajan aseellisesta kukistamisesta, joskin vuoropohjaisuuteen mukautettuna.

RTS- ja TSB-pelien merkittävä eroavaisuus tekoälyn pätevyyden suhteen on TSB-peleissa usein esiintyvät aikarajoitteiden puutteen mahdollistamat strategiset lisäelementit ja niiden myötä noussut strateginen monimutkaisuus. Esimerkiksi Civilization-sarjan peleissä pelaaja hallinnoi armeijan tuotannon ja hallinnan lisäksi monia muitakin olennaisia tekijöitä, kuten yleistä tieteellistä tutkimusta, ruokatuotantoa, terveyttä, uskontoa, taiteita ja diplomaattisia suhteita muihin sivilisaatioihin. Tälläisten strategisten lisäelementtien myötä laadukkaan ja ihmismäiseen strategiseen pelaamiseen kykenevän tekoälyn toteuttamisen haastavuus nousee entisestään. Tästä syystä esim. Civilization-peleissä eksperttitason pelaajat saavat mielekkään haasteen ainoastaan tekoälyn korkeammista haastavuustasoista, jotka on säädetty saamaan valtaisat pelitekniset edut ihmispelaajaan nähden.

3. Tekoälyn kehitysmahdollisuudet strategiapeleissä

Tässä luvussa käydään läpi muutamia malleja ja tekniikoita tekoälyn kehittämiseen strategiapeleissä ja pohditaan niiden soveltuvuutta erilaisiin strategiapelien muotoihin ja osa-alueisiin. Jotkin käsitellyt metodit ovat yleisluontoisempia tekoälyn kehitysmalleja, toiset taas tarkennetumpia ratkaisuehdotuksia tiettyihin tekoälyn puutteisiin joissakin tilanteissa.

3.1. Vastustajamallinnus (opponent modeling)

Vastustajamallinnuksessa pelaaja tai oppiva tekoälytoimija rakentaa sisäistä mallia vastustajistaan. Peleissä, joissa pelin tila on ainoastaan osittain tunnettu, pelaaja joutuu päättelemään pelin tilaa vastustajien toiminnan perusteella. Vastustajamallinnus voidaan jakaa kahteen kategoriaan, implisiittiseen ja eksplisiittiseen mallinnukseen. [Lockett *et al.* 2007]

3.1.1 Implisiittinen vastustajamallinnus

Lockettin ja muiden [2007] mukaan suuri osa vastustajamallinnukseen liittyvästä työstä on keskittynyt nimenomaan implisiittisiin malleihin. Implisiittisessä mallinnuksessa oppimisalgoritmi tallentaa tietoa vastustajistaan omaan sisäiseen representaatioonsa koulutusprosessin sivutuotteena−samalla tavalla kuin ihmispelaaja, jota koulutettaisiin pelauttamalla tätä erilaisia vastustajia vastaan, rakentaisi eräänlaista karkeaa kuvaa vastustajistaan. Implisiittisen mallin tavoitteena on pyrkiä voittamaan tietyt vastustajat, mutta tällaisen implisiittisen tiedon yleistäminen uusiin vastustajiin ilman lisäkoulutusta ei ole selvää. [Lockett *et al.* 2007]

3.1.2 Eksplisiittinen vastustajamallinnus

Eksplisiittisessä vastustajamallinnuksessa toiminnallinen rakenne (tässä tapauksessa pelin neuroverkkotekoäly) koulutetaan ottamaan syötteenä tietoa vastustajasta ja muodostamaan tästä sisäisen mallin, josta se voi päätellä esim. vastustajan pelitapaa luonnehtivia tekijöitä tai odotettavissa olevia strategioita. Eksplisiittisen mallin tarkoitus on siis kyetä rakentamaan yleistyksiä aiemmin tuntemattomistakin vastustajista ilman lisäkoulutusta. [Lockett *et al.* 2007]

Lockett ja muut [2007] ovat kehittäneet *kardinaalivastustajiin* perustuvan neuroverkkoon pohjautuvan eksplisiittisen vastustajamallinnuksen. Tässä mallissa tekoäly muodostaa koulutusvaiheessa vastustajiensa toiminnan perusteella sisäisen mallin joukosta kardinaalivastustajia, jotka muodostavat *vastustajaavaruuden*. Tämän jälkeen tuntemattoman vastustajan toimintaa arvioidaan suhteessa näihin kardinaalivastustajiin ja siihen, missä määrin vastustajan toiminta vastaa kutakin kardinaalivastustajaa. Tällöin vastustajan toimintaa voi-

daan ennakoida kardinaalivastustajien toimintojen perusteella. Lockett ja muut [2007] rakensivat kardinaalivastustajamalliaan Guess It -korttipelissä, jossa vastustajan pelityylin ennakoinnilla on suuri merkitys pelaaja-toimijan omaan peliin, vaikka peli itsessään on melko yksinkertainen. RTS-pelit, kuten Starcraft, ovat pelin mahdollisten tilojen määrässä valtavasti monimutkaisempia, mutta vastustajan pelaajatyypin ja kulloinkin käytössä olevan pelistrategian (esim. nopea "kaikki peliin" -tyylinen hyökkäys) oikeellinen ja oikea-aikainen tunnistus ja vastastrategian omaksuminen on yksi pelin kulun kannalta olennaisimmista tekijöistä ja saattaa mahdollistaa jopa huomattavasti muilta pelitaidoiltaan heikoimman pelaajan voiton. Tästä syystä Lockettin ja muiden [2007] malli tarjoaa mielenkiintoisia mahdollisuuksia myös useiden strategiapelien tekoälyn kehittämiseen.

Strategiapelien, etenkin RTS-pelien kontekstissa tällaisen mallinnuksen hyödyntäminen onnistuneesti on luonnollisesti riippuvainen siitä, että vastustajasta saadaan kerättyä riittävä määrä tietoa. Tätä ongelmaa kuvaillaan ja käsitellään seuraavassa kohdassa.

3.2. Tiedustelu (scouting)

Joitakin poikkeuksia lukuun ottamatta strategiapeleissä, etenkin RTS-peleissä, on lähes aina käytössä ns. "fog of war", eli rajoitettu näkyvyys. Tällöin pelaajien yksiköillä on oma näkökenttänsä, ja kukin pelaaja kykenee näkemään pelialueella ainoastaan sellaisen toiminnan, joka tapahtuu jonkin hänen omistamansa yksikön (tai rakennuksen) näkökenttäalueella. Usein tämä merkitsee sitä, että suuri osa vastustajan toiminnasta on näkymätöntä pelaajalle, ellei tämä pyri aktiivisesti tiedustelemaan vastustajan toimia käyttäen omia yksiköitään. Lisäksi pelialueen maasto on usein pelaajalle tuntematonta, ellei hän tutki sitä yksiköillään, tai kokeneen pelaajan tapauksessa tunne kyseessä olevan pelialueen (kartan) maastoa ulkoa.

3.2.1 Maaston tiedustelu

Sin ja muiden [2014] mukaan strategiapelien tekoälytoteutuksessa kartan rakenteen tiedustelun tarve usein kierretään antamalla tekoälylle suoraan tiedot pelattavan kartan maastosta. Tällainen "tekoälyn huijaus" kuitenkin vaikuttaa negatiivisesti etenkin noviisipelaajien pelikokemukseen. Jotta tekoälyn voidaan katsoa toimivan tasaveroisena ihmispelaajan korvikkeena, on sen kyettävä suorittamaan tiedustelu ilman ennakkotietoja kartan rakenteesta.

Maastontiedustelussa olennaista on tiedusteluun käytettävän yksikön oikeellinen hallinta siten, että tarvittava data kyetään keräämään mahdollisimman lyhyessä ajassa välttäen tiedustelijayksikön vaurioitumista ja tuhoutumis-

ta. Si ja muut [2014] ovat kehittäneet RTS-peleihin sovellettavan maastonkartoitus-tiedustelualgoritmin, joka perustuu maastontutkimus-robotiikan tutkimukseen. Monissa RTS-peleissä, kuten Starcraftissa, alkupelin tiedustelu suoritetaan pelin alussa käytettävissä olevalla, edullisella ja Starcraft-pelin tapauksessa lähestulkoon taistelukyvyttömällä rakentaja/työläis-yksiköllä, joka tukikohdan rakentamisen tai resurssien keräämisen sijasta lähetetään tiedustelemaan maastoa tai vihollisen toimintaa. Sin ja muiden [2014] algoritmi perustuu yhden tällaisen tiedustelijaksi määritellyn työläisyksikön hallintaan, ja noudattaa karkeasti seuraavaa kaavaa:

A) *Tiedusteluyksikkö havainnoi näkökenttänsä ympäristön*
B) *Havainnoidut karttaelementit integroidaan karttaa edustaviin järjestelmiin*
C) *Seuraavat potentiaaliset sijainnit määritellään karttaa edustavista järjestelmistä määrittelystrategian mukaan*
D) *Potentiaaliset sijainnit arvioidaan monilla eri kriteereillä*
E) *Optimaalinen seuraava sijainti valitaan*
F) *Yksikkö siirtyy valittuun sijaintiin ja aloittaa kohdasta a)*

Algoritmin toiminnan kannalta olennaisimpana voidaan pitää kohtaa D, jossa sovelletaan usean kriteerin päätöksentekoa (multiple-criteria decision-making, MCDM). Toisin kuin useissa perinteisissä reitinmäärittelyongelmissa, tiedusteluyksikön tarkoitus ei ole ainoastaan etsiä optimaalisinta reittiä kahden pisteen välillä, vaan hoitaa tiedusteluprosessi koko alueen suhteen. Tällöin on jatkuvasti määriteltävä tiedustelijayksikön seuraava sijainti (next-best-view, NBV) siten, että siihen siirtymällä on odotettavissa mahdollisimman suuri määrä hyödyllistä tietoa ympäristöstä. NBV-ehdokkaita arvioidaan useilla kriteereillä – maastonmuotojen lisäksi RTS-pelien kontekstissa on arvioinnissa otettava huomioon mahdollinen tiedustelijaa uhkaavien vihollisyksikköjen läsnäolo sekä muut pelin kannalta olennaiset karttaelementit. [Si *et al.* 2014]

Useissa rajoittuneen näkyvyyden strategiapeleissä, kuten Starcraftissa, pelaajatoimijoiden aloituspaikat tai ainakin niiden mahdollinen joukko tunnetaan ennalta, jolloin maaston kattava tiedustelu ei ole sinällään välttämätöntä vastustajan tukikohdan löytämiseksi. Niissä peleissä, jossa näin ei ole, maaston kartoituksen tärkeys kuitenkin korostuu, sillä vastustajan tukikohdan (ja muunkin mahdollisen toiminnan) nopea paikantaminen on edellytys seuraavassa alakohdassa käsiteltävälle tiedustelun muodolle, vastustajan strategian tiedustelulle.

3.2.2 Vastustajan strategian tiedustelu

Vastustajan toiminnan selvittäminen on ehkä maaston kartoitustakin tärkeämpi tiedustelun aspekti strategiapeleissä, etenkin korkealla tasolla pelatessa. Tällaisen tiedustelun tärkeys korostuu Starcraft-strategiapelissä, sillä vastustajan strategian ennakointi ja tämän toimiin perustuva oikea-aikainen reagointi vaativat tietoa tämän toiminnasta. Voidakseen omaksua voittoisan strategian, pelaajan on kyettävä lähettämään tiedusteluyksikkö vihollisen tukikohtaan ja ennakoitava tämän strategiaa etenkin alkupelin suhteen keräämistään puutteellisista tiedoista. Tiedustelijayksikkö kykenee kuitenkin havainnoimaan vain välitöntä ympäristöään, jolloin sen on kyettävä liikkumaan vastustajan tukikohdassa älykkäästi paljastaakseen olennaista tietoa. Lisäksi vastustaja kykenee reagoimaan tiedustelijayksikön saapumiseen pyrkimällä tuhoamaan sen tai viivyttämään omaa strategisesti paljastavaa toimintaansa kunnes tiedustelija on tuhottu. [Park *et al*. 2012]

Park ja muut [2012] ovat soveltaneet tiedustelu- ja koneoppimisalgoritmeja kehittämäänsä Starfcraft-tekoälykilpailuissa menestyneeseen Xelnaga-nimiseen tekoälytoimijaan (bottiin). Toisin kuin monet muut oppivien Starcraft-tekoälytoimijoiden suunnittelijat, he eivät kouluttaneet bottiaan saatavilla olevien korkeatasoisten pelinauhoitusten avulla, vaan sen sijaan pelauttivat tätä aitoja ihmispelaajia vastaan. Kyseisissä peleissä he hyödynsivät kehittämiään tiedustelualgoritmeja.

Haastavin osuus vastustajan toiminnan tiedustelussa on algoritmin toteuttaminen niin, että tiedustelijayksikkö liikkuu tarpeeksi lähellä vihollisen toimintaa paljastaakseen sen pelaajille, mutta ei niin lähellä, että sen kimppuun olisi helppoa hyökätä alkupelissä käytettävissä olevilla sotajoukoilla tai työläisyksiköillä. Park ja muut [2012] toteuttivat algoritminsa niin, että tiedustelijayksikön toiminta matkii yleistä ihmispelaajan tiedustelutoimintaa. Yksikkö kiertää vastustajan rakennuksia näkömatkansa etäisyydellä ja kerää tietoa strategian kannalta paljastavasta alkupelin rakennustoiminnasta niin pitkään, kunnes vastustaja onnistuneesti tuhoaa tiedustelijan. Monissa tapauksissa tiedustelija ehtii operoida vastustajan tukikohdassa riittävän pitkään ennen tuhoutumistaan, jotta tiedustelijan keräämistä vastustajan rakennusjärjestystiedoista ajoituksineen voidaan päätellä vastustajan alkupelin strategiaa ja varautua siihen. [Park *et al.* 2012]

3.3. Bayesilainen todennäköisyysmallinnus tekoälylle

Tekoälyn luomiseen liittyy yleisesti kaksi ongelmaa: nykytilan havainnointi ja päätöksen teko. Havainnointia RTS-peleissä rajoittaa yleisimmin aiemmin kohdassa 3.2 käsitelty näkyvyyden puutteellisuus. Lisäksi pelitilanteen ja pelin osa-

alueiden abstraktointi tekoälyn käsiteltäväksi aiheuttaa tiedon epävarmuutta päätöksien teossa. [Synnaeve *et al.* 2015]

Synnaeven ja muiden [2015] mukaan bayesilaiseen todennäköisyyslaskentaan perustuva mallinnus on yksi mahdollinen tapa käsitellä RTS-pelien monimutkaisuutta tekoälyn toteutuksessa. He kehittivät bayesilaiseen todennäköisyyslaskentaan perustuvat tekoälymallin, joka kykenee käsittelemään tehokkaasti lähes kaikille strategiapeleille ominaista tiedon epävarmuutta sekä abstraktoimaan mielekkäällä tavalla pelisuorituksen eri tasot, strategian, taktiikan sekä mikrohallinnan toisiinsa liittyvällä tavalla. Synnaeven ja muiden [2015] mukaan bayesilainen mallinnus mahdollistaa tekoälyn päätöksenteon siten, että huomioon otetaan koko tilastollinen jakauma, mikä edesauttaa riskiarviointia.

Pelaajan mahdollisten päätösten määrä kullakin hetkellä on Starcraftin tapaisissa RTS-peleissä huomattavasti laajempi kuin esimerkiksi shakissa. Karsiakseen tätä vaihtoehtojen joukkoa, Synnaeven ja muiden [2015] malli noudattaa seuraavien mahdollisten päätösten rajaamiseen kaksiulotteista hierarkiaa. Taktiikkatasolla päätökset ovat sitä todennäköisempiä, mitä paremmin ne vastaavat strategiatason päätöstä, ja samalla tavalla mikrohallintatason päätökset ovat alisteisia taktiikkatasolle. Lisäksi kullakin kolmesta päätöstasosta ajan hetken *t* mahdollista toimintojoukkoa rajaa ajan hetkellä *t-1* tehty päätös [Synnaeve *et al.* 2015]. Esimerkkinä voitaisiin kuvailla tilannetta, jossa strategiatason päätös *aggressiivisesta* strategiasta lisää taktiikkatasolla *suoran hyökkäyksen* päätöksen todennäköisyyttä, jota taas taktiikkatason sisällä ajallisesti seuraa todennäköisemmin *iske-ja-pakene* -päätös kuin esimerkiksi *huomaamaton tunkeutuminen*.

3.4 Oppiva tekoäly

Monissa strategiapeleissä, etenkin Starcraftin kaltaisissa kilpailu- ja turnauskeskeisissä RTS-peleissä, on mahdollista nauhoittaa, julkaista ja jakaa pelien toisintoja (replays). Etenkin Starcraft-pelin suosio on johtanut siihen, että tällaisia pelinauhoituksia on runsaasti saatavilla, ja niissä esiintyy laaja joukko strategioita ja pelityylejä [Weber *et al.* 2009]. Toisintojen laaja saatavuus mahdollistaa oppivan tekoälyn kouluttamisen ja kohdassa 3.1 käsitellyn vastustajamallinnuksen esim. ammattilaistason pelien toisinnoista saatavan datan avulla. Samaan tapaan koneoppimisalgoritmeja hyödyntävä tekoälytoimija voidaan toteuttaa omista peleistään oppivaksi, kuten Parkin ja muiden [2012] Xelnagabotti. Koneoppimisen voidaankin katsoa tarjoavan erittäin paljon kehitysmahdollisuuksia haastavan strategiapeli-tekoälyn kehittämiseen integroituna yhteen muiden tekniikoiden kanssa.

Weber ja muut [2009] ovat kehittäneet tiedonlouhintaan ja koneoppimiseen perustuvan tekniikan vastustajan strategian ennakointiin. He rakensivat web-agentin, joka etsii internetistä suosituilta pelitoisinto-sivustoilta suuren määrän sekä ammattilaistason että korkean tason amatööripelien nauhoituksia, ja muuntaa ne pelitoimintakirjauksiksi. Näistä kirjauksista rakennetaan kullekin pelin osallistujalle piirrevektori, joka luokitellaan tiettyyn strategiaan kuuluvaksi. Koneoppimisalgoritmeilla pyritään tämän jälkeen havaitsemaan ja luokittelemaan vastustajan strategia ja ennakoimaan sen perusteella tämän toimintoja.

Weberin ja muiden [2009] pyrkimys oli rakentaa sellainen yleinen malli eksperttitason StarCraft-pelaamisesta, joka ei perustu yksittäisten vastustajien mallintamiseen tai ainoastaan muutamiin satoihin pelinauhoituksiin. He suorittivat mallillaan useita eri kokeita, joista osassa myös todellisille peleille ominaista rajoitettua näkyvyyttä ja siitä johtuvaa puutteellista tietoa simuloitiin mm. "häiriösignaalilla" vastustajien toiminnan suhteen. Heidän mallinsa kykeni ennustamaan sekä vastustajan strategiaa että ajoitusta keskimäärin verrokkeja paremmin [Weber *et al.* 2007].

4. Yhteenveto ja loppupohdinnat

Strategiapelien tekoäly on e-urheilun nousun myötä saanut osakseen paljon aiempaa enemmän akateemistakin kiinnostusta. Lisäksi monet strategiapelin tekoälyn kehityksessä kohdattavat haasteet vastaavat osittain muillakin tekoälyä hyödyntävien tutkimusalueiden saralla kohdattavia, kuten esimerkiksi pelialueen kartoitus RTS-pelissä ja ympäröivän maaston kartoitus robotiikassa [Si *et al.* 2014]. Vastustajien toiminnan mallinnus ja ennakointi koneoppimistekniikoilla puolestaan tarjoaa stragegiapelien lisäksi paljon kehitysmahdollisuuksia myös muiden peligenrejen tekoälykehitykselle.

Monet tässä katsauksessa käsitellyistä tekoälynparannusmenetelmistä on kehitetty Starcraft tai Starcraft II -pelien pohjalta, mutta ovat luultavammin peligenren keskinäisen pelimekaanisen yhdenmukaisuuden vuoksi useimmiten sinällään yleistettävissä lähes minkä tahansa RTS-pelin tekoälyn toimintaan. StarCraft-pelin hallitsevuutta pelitekoälytutkimuksessa selittää osaltaan myös se, että StarCraft-pelissä toimiville epävirallisille tekoälyboteille on järjestetty avoimia kilpailuja pelitekoälykonferensseissa (IEEE CIG ja AAAI AIIDE).

Yhdistelemällä, jatkokehittämällä ja soveltamalla käytäntöön tässä katsauksessa esitettyjen tekniikoiden kaltaisia tekoälynkehitysmetodeja saattaisi olla mahdollista luoda huomattavasti nykypäivän kaupallisten pelien tasoa älykkäämpiä ja ihmismäisempiä tekoälytoimijoita strategiapeleihin. Erityisen mielenkiintoisena tulevaisuuden kehityksen kannalta voidaan pitää etenkin oppi-

via tekoälytoimijoita. Monissa koneoppimista soveltavissa tekniikoissa hyödynnetään laajalti saatavilla olevia pelitoisintoja tai kehitetään toimijoita, jotka kykenevät joko omatoimisesti tai opastetusti oppimaan peleistä, joita ne itse pelaavat. Nykyään suuri osa modernien strategiapelien ottelutoiminnasta organisoidaan ja toteutetaan keskitetysti kunkin pelin julkaisijan omilla julkaisualustoilla/palvelinjärjestelmillä (kuten Blizzard Entertainment -yhtiön Battle.Net). Mikäli tämänkaltaisiin palvelinjärjestelmiin ja niiden sisällä toimiviin peleihin integroitaisiin oppivaa tekoälytoimintaa, olisi sillä jatkuvasti käytössään suuri ja erilaisista ihmispelaajista koostuva vastustajajoukko sekä valtava määrä pelidataa. Tällainen tekoälyjärjestelmä saattaisi lopulta kehittyä potentiaalisilta pelitaidoiltaan eksperttipelaajien tasolle (tai jopa niiden yli) sekä jopa pysyä ihmispelaajien tavoin mukana ns. "metapelissä", jolla tarkoitetaan pelaajien keskuudessa kulloinkin pinnalla olevia strategioita ja niiden vastastrategioita. Tällöin olisi mahdollista, että ammattitason pelaajatkaan eivät enää kykenisi pelisuoritusten perusteella erottamaan tekoälypelaajaa vertaisestaan.

Viiteluettelo

Andrew Chiou. 2007. A game AI production shell framework: generating AI opponents for geomorphic-isometric strategy games via modeling of expert player intuition. *DIMEA '07 Proceedings of the 2nd International Conference on Digital Interactive Media in Entertainment and Arts*, 84-90.

Quentin Gemine, Firas Safadi, Raphael Fonteneau and Damien Erns. 2012. Imitative learning for real-time strategy games. In: *2012 IEEE Conference on Computational Intelligence and Games,* 424-429.

Alan J. Lockett, Charles L. Chen and Risto Miikkulainen. 2007. Evolving explicit opponent models in game playing. In: *GECCO '07 Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation*, 7-12.

Hyunsoo Park, Ho-Chul Cho, KwangYeol Lee and Kyung-Joong Kim. 2012. Prediction of early stage opponents strategy for StarCraft AI using scouting and machine learning. In: *WASA '12 Proceedings of the Workshop at SIGGRAPH Asia*, 7-12.

Chen Si, Yusuf Pisan, Chek Tien Tan. 2014. A Scouting Strategy for Real-Time Strategy Games. In: *IE2014 Proceedings of the 2014 Conference on Interactive Entertainment,* 1-8.

Gabriel Synnaeve and Pierre Bessière. 2015. Multi-scale Bayesian modeling for RTS games: an application to StarCraft AI. To appear in *IEEE Transactions on Computational intelligence and AI in Games*.

Ben G. Weber and Michael Mateas. 2009. A Data Mining Approach to Strategy Prediction. In: *Proc. of the IEEE Symposium on Computational Intelligence and Games,* 140-147.

Käsinkirjoitettujen numeroiden tunnistus koneoppimisalgoritmeilla

Tuomas Taubert

Tiivistelmä

Visuaalisen informaation ymmärtäminen on edelleen tietokoneille erittäin haastava tehtävä. Koneoppimisalgoritmeilla on jo saavutettu hyviä tuloksia monissa erilaisissa tunnistustehtävissä, mutta kaikkein vaikeimmissa tehtävissä ihmiset kykenevät vielä parempiin suorituksiin. Esittelen tässä tutkielmassa erilaisia koneoppimisalgoritmeja, ja tutkin millaista hyötyä on mahdollista saavuttaa yhdistelemällä näitä algoritmeja suuremmiksi kokonaisuuksiksi.

Avainsanat ja -sanonnat: koneoppiminen, lajittelu, numeroiden tunnistus, kNN, SVM, neuroverkko, yhdistyt algoritmit

1. Johdanto

Lukeminen on hyvin monimutkainen tehtävä. Ihminen suoriutuu merkkien tunnistamisesta pitkälle kehittyneen näköaivokuoren avulla, joka harjaantuneesti tulkitsee näköaistin tuottamasta informaatiosta tarvittavan tiedon. Tietokoneelle tehtävästä suoriutuminen saattaa tuottaa suuria vaikeuksia, jopa siinä määrin, että tekstintunnistus (CAPTCHA) on yksi hyvin yleinen tapa pyrkiä erottelemaan ihmiset tietokoneista [Korakakis *et al.* 2014].

Yksi lähestymistapa tekstintunnistuksen ongelmaan on koneoppiminen, ja käsialan tulkinta onkin klassinen esimerkki ohjatusta koneoppimisesta. Ohjatussa koneoppimisessa käytetään algoritmeja, joille voidaan opettaa merkkien tunnistusta valmistellulla datalla. Tällöin algoritmille on valmiiksi luokiteltu kuvia niiden sisältämien merkkien mukaan, jolloin algoritmi voi ”oppia” annetusta datasta, minkälaisia piirteitä kuhunkin merkkiin kuuluu. Kun algoritmia on opetettu tarpeeksi suurella määrällä opetusdataa, sen tarkoituksena on pystyä yleistämään oppimansa annettavaan testiaineistoon ja päättelemään, mistä merkeistä testiaineistossa on kyse.

Tässä tutkielmassa vertaillaan erilaisia koneoppimista hyödyntäviä algoritmeja käsinkirjoitettujen numeroiden tunnistuksessa ja tutkitaan, onko useamman erilaisen verkon tai algoritmin yhdistämisestä merkittävää hyötyä tunnistuksessa. Aluksi esitellään lyhyesti käsinkirjoitetun tekstin tunnistukseen liittyviä vaiheita. Luvussa 3 kuvataan erilaisia ongelmaan kehitettyjä koneoppimisalgoritmeja ja luvussa 4 tutkitaan, kuinka algoritmien yhdistäminen vaikuttaa tunnistustulokseen.

2. Käsinkirjoitettujen numeroiden tunnistus

Käsinkirjoitus on modernissakin yhteiskunnassa säilyttänyt paikkansa useilla eri aloilla, esimerkiksi kirjeiden ja korttien osoitteet kirjoitetaan usein käsin. Tämä on myös hyvä esimerkki ongelmasta, jossa voidaan hyödyntää koneoppimisalgoritmeja numeroiden tunnistamisessa, ja näin tehostaa postin lajittelujärjestelmän toimintaa. Kyseiseen ongelmaan on haettu koneoppimisesta ratkaisua jo pitkään, esimerkiksi Matan ja muut [1992] ovat kehittäneet neuroverkon ratkaisuksi USA:n postin ZIP-koodien tunnistukseen jo 1990-luvun alussa. Tietokoneiden laskentatehon lisääntyessä entistä monimutkaisempienkin ratkaisujen toteutus on tullut mahdolliseksi, ja siksi ratkaisujen kehitys jatkuu edelleen.

Itse numeroiden tunnistus voidaan jakaa kolmeen alavaiheeseen: *esiprosessointi*, *piirreirrotus* sekä *lajittelu* [Lauer *et al.* 2007]. Näitä vaiheita voi edeltää tarvittaessa aiempi esiprosessointivaihe sekä segmentointi, jos tarkoituksena on tulkita numerojonoa. Segmentoinnissa numerojonosta erotellaan yksittäiset numerot, jonka jälkeen päästään varsinaiseen tunnistusvaiheeseen. Esiprosessointi ja piirreirrotus ovat usein välttämättömiä lajittelun onnistumiselle, mutta tässä tutkielmassa ne käydään läpi vain lyhyesti ja keskitytään tarkemmin lajitteluvaiheeseen.

2.1. Esiprosessointi

Esiprosessoinnin tavoitteena on erottaa varsinainen signaali kohinasta, ja siten parantaa seuraavien vaiheiden tarkkuutta. Käsinkirjoitetussa tekstissä kohina tarkoittaa fyysisen lian lisäksi vaihtelua eri henkilöiden käsialassa: numeroiden koko, vinous, viivan paksuus ja monet muut ominaisuudet ovat eri kirjoittajilla täysin erilaisia. [Pesch *et al.* 2012] Esiprosessoinnin tehtävänä on siis yrittää poistaa näiden tekijöiden ylimääräinen vaikutus ja muuttaa numerot normalisoituun muotoon. Mahdollisia esiprosessointitapoja ovat esimerkiksi seuraavat:

- Skaalaus – kuva numerosta skaalataan standardoituun kokoon.
- Vinouden korjaus – numero suoristetaan tarvittaessa esimerkiksi kursiivista suoraksi.
- Kohinan poisto – korjataan kuvan ottamisesta syntynyttä kuvanlaadun heikentymää.
- Kavennus – viivan paksuus normalisoidaan 1 pikselin levyiseksi
- Värikorjaukset / Harmaaskaala - kuvan kontrastia voidaan säätää tai muuttaa se esimerkiksi mustavalkokuvaksi.

2.2. Piirreirrotus

Piirreirrotuksen tavoitteena on mitata lähteestä erilaisia ominaisuuksia, joiden perusteella lajitteluvaiheessa on mahdollista tunnistaa, mistä numerosta on kyse. Yksinkertaisimmillaan piirreirrotuksessa esiprosessoidulle kuvalle ei tehdä mitään, vaan lajittelualgoritmille välittyy pelkästään esiprosessoitu kuva. Piirreirrotuksen tarkoituksena on kuitenkin tuottaa lajittelun kannalta hyödyllistä ylimääräistä informaatiota, jotta numeron lajittelu olisi mahdollista. Esimerkiksi seuraavia piirteitä voidaan yrittää mitata [Matan *et al.* 1992]:

- reunojen sijainnit
- viivojen päätekohdat
- viivojen risteyskohdat
- suljetut silmukat.

Caesar ja muut [1993] ehdottavat, että kuva voitaisiin esimerkiksi jakaa vaakasuunnassa viiteen osaan ja suorittaa piirreirrotusta eri osa-alueille: ylimmästä osasta löytyvät vaakaviivat liittyvät numeroihin viisi ja seitsemän, kun taas alimmasta osasta löytyvät silmukat viittaavat numeroihin kuusi ja kahdeksan. Oikein tehtynä piirreirrotuksella on mahdollista tuoda runsaasti lisäinformaatiota lajittelualgoritmille, kun taas heikosti toteutettu piirreirrotus voi tuoda enemmän ristiriitaista kuin hyödyllistä informaatiota.

2.3. Lajittelu

Lajitteluvaiheessa algoritmit käyttävät aiempien vaiheiden tuottamaa dataa ja antavat sen perusteella arvion siitä, mikä numero on kyseessä. Riippuen toteutustavasta, algoritmi voi arvioida luokitellun luokan lisäksi esimerkiksi kuinka suurella todennäköisyydellä kyseinen näyte kuuluu tiettyyn luokkaan.

3. Lajittelualgoritmit

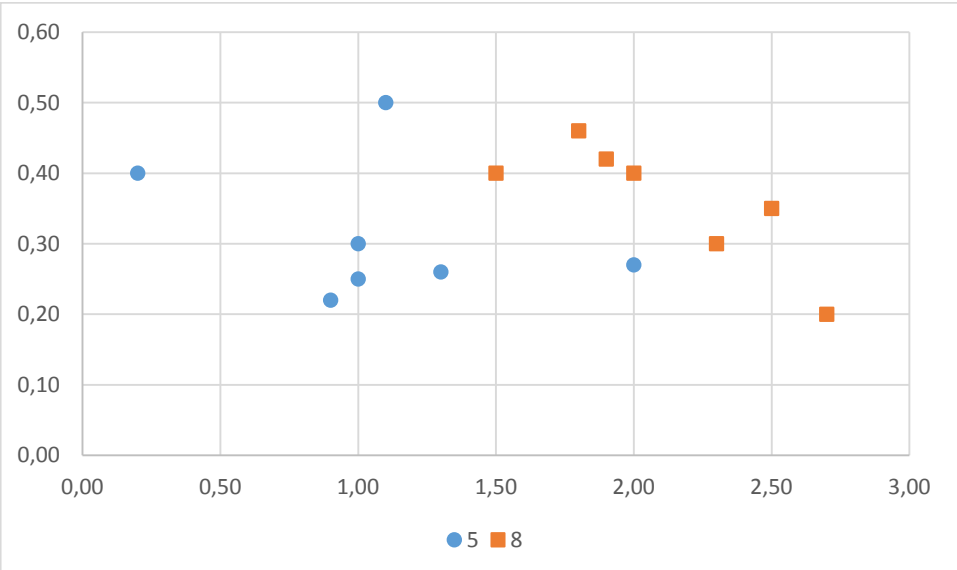
Koneoppiminen pitää sisällään lukemattomia erilaisia lajittelualgoritmeja ja niiden erilaisia variaatioita, mutta tähän tutkielmaan on valittu seuraavat algoritmit Liun ja Fujisawan [2005] esittämän jaottelun mukaisesti: *k-nearest neighbors -klusterointialgoritmi*, *tukivektorikone* sekä *neuroverkot*. Liu ja Fujisawa [2005] jakavat lajittelualgoritmit artikkelissaan tilastotieteellisiin metodeihin, kernelimetodeihin sekä neuroverkkoihin, joihin edellä mainitut algoritmit kuuluvat samassa järjestyksessä. Näillä algoritmeilla on saavutettu hyviä tuloksia esimerkiksi MNIST-tietokannan kanssa [LeCun 2016; Liu and Fujisawa 2005], joka esiintyy useimmissa lähteissä. MNIST-tietokanta koostuu 60 000 käsinkirjoitetun numeron harjoitusjoukosta ja 10 000 numeron testijoukosta, ja eri luokittelijoilla on päästy tällä aineistolla alle 1 % virhemäärään [LeCun 2016].

3.1. K-nearest neighbors

K-nearest neighbors (kNN) -klusterointialgoritmi on yksi vanhimmista ja yksinkertaisimmista lajittelualgoritmeista [Weinberger *et al.* 2005] ja siksi hyvä lähtökohta tutkittaessa erilaisia vaihtoehtoisia lajittelualgoritmeja. KNN-algoritmissa opetusjoukon esimerkit ovat moniulotteisia vektoreita ja testijoukon yksilöitä verrataan niihin etsien mahdollisimman samanlaisia esiintymiä testijoukosta.

3.1.1. Oppimisvaihe

Yksinkertaisimmillaan kNN-algoritmi ei vaadi oppimisvaiheessa suoritettavia laskutoimituksia, vaan oppimisvaiheeseen kuuluu vain harjoitusjoukon piirrevektorien ja luokkien tallennus. Piirrevektoreilla tarkoitetaan tässä yhteydessä kaikkia niitä ominaisuuksia, jotka piirreirrotusvaiheessa on mitattu harjoitusjoukon jäsenistä. Yksinkertaisimmillaan mitattuja ominaisuuksia on vain kaksi, jolloin piirrevektoreita voidaan havainnollistaa kaksiulotteisella kaaviokuvalla. (Kuva 1). Kuvan esimerkissä on harjoitusjoukon lukujen 5 ja 8 mitatut ominaisuudet, kuvan tummuus y-akselilla ja pisimmän yhtenäisen viivan pituus x-akselilla. Käytännössä mitattavia ominaisuuksia on huomattavasti enemmän, jolloin visuaalinen havainnollistus ei enää onnistu, mutta kNN-algoritmi voi edelleen etsiä kohteen lähimmät naapurit.

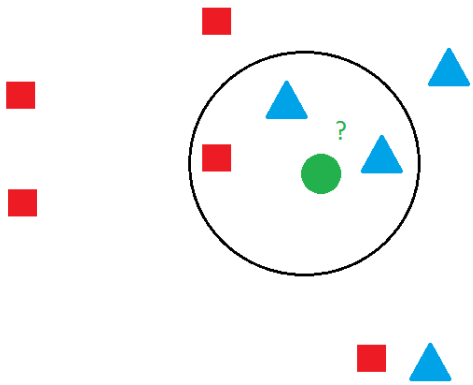


Kuva 1: Kaksiulotteiset piirrevektorit havainnollistettuna. Harjoitusjoukon lukujen 5 ja 8 ominaisuudet.

3.1.2. Testausvaihe

Testivaiheessa testausjoukon alkiolle suoritetaan normaaliin tapaan sama piirreirrotus kuin opetusjoukon alkiolle. Tämän jälkeen algoritmi etsii *k* ominaisuuksiltaan lähintä opetusjoukon alkiota. Tutkittavan alkion luokka määräytyy

näiden *k*:n lähimmän alkion äänestyksen perusteella. *K*:n arvo voi olla määriteltynä jo algoritmissa valmiina, tai optimaalisin *k*:n arvo voidaan määritellä opetusdatan perusteella [Babu *et al.* 2014]. Kuvassa 2 on havainnollistettuna kuinka äänestys toimii: jos esimerkiksi on valittu *k*=3, algoritmi etsii testattavan alkion piirrevektoria lähinnä olevat kolme piirrevektoria. Niiden määritetyt luokat otetaan huomioon, ja esimerkin tapauksessa alkio määritettäisiin kolmioksi, sillä kolmioita on enemmistö etsityssä kolmen piirrevektorin joukossa.



Kuva 2: Opetusjoukon piirrevektoreita (neliö ja kolmio), sekä testattavan alkion piirrevektori (ympyrä). Alkiot jotka osallistuvat äänestykseen kun *k*=3 ovat ympyröitynä.

Etäisyys tutkittavasta alkiosta opetusdatan alkioihin määritetään usein euklidisen etäisyyden avulla, mutta Weinberger ja muut [2005] nostavat esiin tähän liittyviä ongelmia: jos esimerkiksi piirreirrotuksessa määriteltyjen ominaisuuksien skaala on hyvin erilainen, tai niissä esiintyviä eroja on hankala vertailla, pelkän Euklidisen etäisyyden laskeminen ei tuota haluttua tulosta. Ratkaisuna tähän voi olla esimerkiksi piirteiden skaalaus, niin että eri piirteissä esiintyvät erot vaikuttavat etäisyyteen yhtä paljon.

Tähän saakka kuvattu algoritmi on hyvin pelkistetty versio kNN-algoritmista, sillä algoritmiin on kehitetty monenlaisia suorituskykyä ja tunnistustarkkuutta parantavia tekniikoita, joilla kNN-algoritmi saadaan toimimaan muiden tässä tutkielmassa kuvattavien algoritmien tasolla, ja tietyissä tilanteissa jopa paremmin. Esimerkiksi Weinberger ja muut [2005] kuvaavat useita tapoja, joilla algoritmia voi parannella.

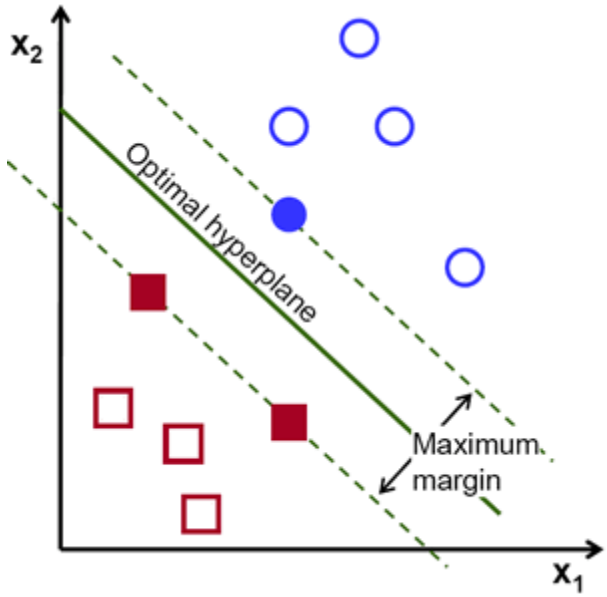
3.2. Tukivektorikone

Tukivektorikone (Support Vector Machine, SVM) on alun perin 1990-luvulla kehitetty lajittelualgoritmi, joka pyrkii etsimään hypertason, jolla pystytään erottamaan yksittäisten luokkien jäsenet toisistaan. Hypertasolla tarkoitetaan osa-aluetta, jolla on yksi ulottuvuus vähemmän kuin ympäröivällä tilalla. Esi-

merkiksi 3-ulotteisessa avaruudessa hypertaso on taso, kun taas 2-ulotteisessa avaruudessa hypertaso on suora. [Schölkopf *et al.* 1997]

3.2.1. Oppimisvaihe

Yksinkertaisimmillaan tukivektorikone toimii eroteltaessa kahta luokkaa kahden ominaisuuden perusteella toisistaan. Esimerkkinä piirreirrotuksessa on jälleen tutkittu kahta piirrettä x_1 ja x_2. Näitä piirteitä eri alkioilla voidaan nyt havainnollistaa 2-ulotteisessa kaaviossa (Kuva 3). Tukivektorikone pyrkii sovittamaan alkioiden välille hypertason siten, että eri luokkien ja hypertason väliin jäävä marginaali on mahdollisimman suuri.

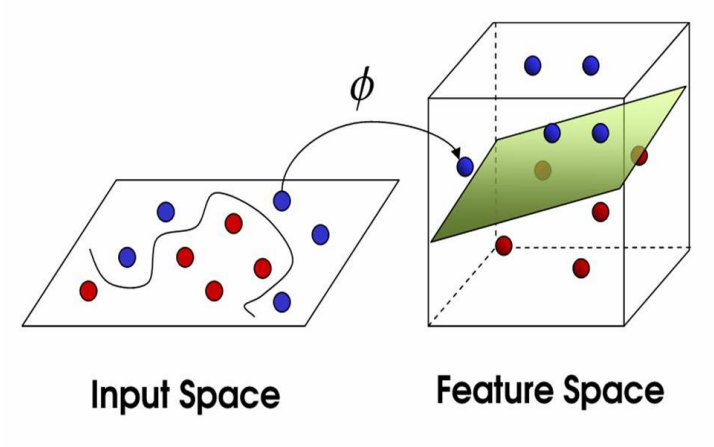


Kuva 3: Tukivektorikoneen kahden eri luokan alkioiden väliin sijoittama hypertaso

Jos opetusdatan luokat ovat erotettavissa toisistaan lineaarisesti, voidaan määritellä kaksi samansuuntaista hypertasoa, jotka erottavat luokat toisistaan ja joiden välinen matka on mahdollisimman suuri. Kuvassa 3 näitä hypertasoja kuvaavat katkoviivat, ja ne voidaan määritellä sellaisena joukkona, jonka alkioille $\vec{x}$ pätee $\vec{w} \cdot \vec{x} - b = 0$, missä $\vec{w}$ on normaalivektori kohtisuoraan hypertasoa kohti ja $\frac{b}{||\vec{w}||}$ on kohtisuora etäisyys hypertasosta keskipisteeseen. Kun nämä marginaalit ovat määriteltynä, voidaan optimaalisen hypertaso määrittää luokkien väliin siten, että se on yhtä kaukana molemmista marginaaleista. [Tuba and Bacanin 2015]

Kuvassa 3 hypertasoksi on piirretty suora, joka on helppo ratkaisu kyseiseen ongelmaan. Todellisuudessa ratkaisu on harvoin lineaarinen, esimerkiksi kuvan 1 tapauksessa suoraa luokkien välille ei ole mahdollista piirtää. Siinä tapauksessa luokkien erotus toisistaan voidaan suorittaa kernel-funktiota käyt-

tämällä. Tällöin yllämainitun hypertason kaavan pistetulo korvataan kernel-funktiolla, ja opetusdata projisoidaan korkeammalle ulottuvuusavaruudelle, tavoitteena tehdä mahdolliseksi luokkien erottamisen hypertasolla [Tuba and Bacanin 2015]. Kuvassa 4 on esitetty, kuinka muuttamalla kaksiulotteinen kaavio kolmiulotteiseksi, on mahdollista löytää hypertaso, jolla luokat saadaan erotettua toisistaan.



Kuva 4: Kernel-funktion toiminta.

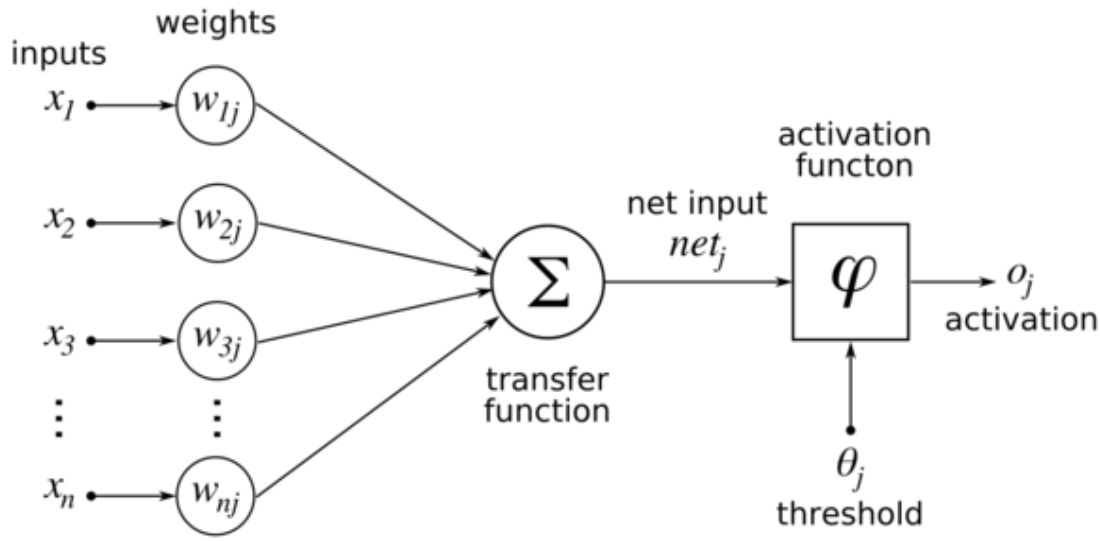
3.2.2. Testausvaihe

Numeroiden tunnistuksessa tukivektorikoneella on vielä eräs ongelma. Tukivektorikone toimii parhaiten eroteltaessa kahta luokkaa toisistaan. Numeroiden tunnistuksessa luokkia on kuitenkin 10 (numerot 0-9), joten luokkien määrää tulee vähentää, jotta algoritmi voi toimia halutulla tavalla. Tähän ongelmaan on olemassa useita ratkaisuja: Tuba ja Bacanin [2015] esittävät kaksi ratkaisua, jotka hyödyntävät useita SVM-algoritmeja. Ensimmäinen vaihtoehto on *yksi kaikkia vastaan*, jossa jokaiselle numerolle luodaan oma SVM, jossa luokkia on kaksi: kyseisen numeron alkiot sekä muiden numeroiden joukko. Tällöin on tarpeen luoda 10 eri SVM:ia, jotka vertaavat, kuuluuko tutkittava alkio kyseisen luokan alkioihin vai ”muihin”. Toinen vaihtoehto on *yksi yhtä vastaan*, jolloin luodaan jokaista numeroparia kohden oma SVM:n. Tällöin tarvitaan 45 erilaista SVM:ia, sekä esimerkiksi sääntöperustainen logiikka, jolla näin monen SVM:n tuloksista saadaan pääteltyä mistä numerosta on kyse. Näiden ratkaisujen lisäksi on mahdollista toteuttaa esimerkiksi jollain muulla lajittelualgoritmilla esivalinta, joka arvioi mihin kahteen luokkaan tutkittava alkio vaikuttaa kuuluvan, ja sen perusteella käytetään sopivaa yksi yhtä vastaan -SVM:ia.

3.3. Neuroverkot

Neuroverkoilla tarkoitetaan joukkoa algoritmeja, joiden esikuvana on alun perin ollut ihmisen keskushermoston toiminta. Neuroverkkojen perustana ovat yksittäiset neuronit, joihin saapuu syötteitä aiemmilta neuroneilta, ja jotka rea-

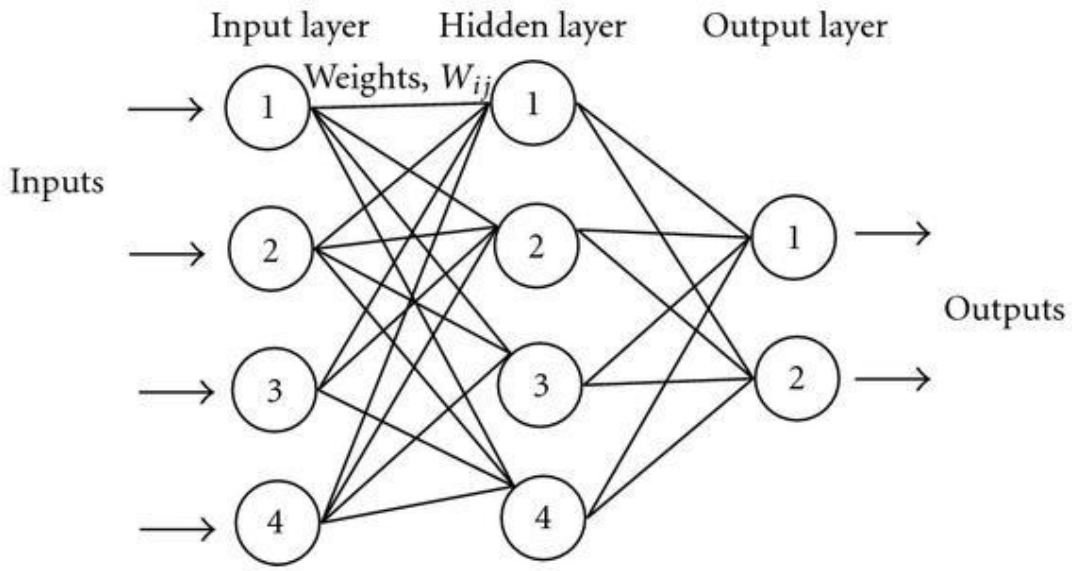
goivat saamaansa syötteeseen aktivointifunktionsa mukaan joko aktivoitumalla tai pysymällä passiivisena.



Kuva 5: Yksittäisen neuronin toiminta neuroverkossa.

Kuvassa 5 on esitetty yksittäisen neuronin toimintamekanismi. Neuroniin tulee syötteitä (x_1, x_2 … x_n), joihin sovelletaan painokertoimia (w_{1j}, w_{2j} … w_{nj}). Syötteet määräytyvät tutkittavan alkion perusteella, kun taas painot on määritetty oppimisvaiheessa. Neuroni laskee syötteiden ja painojen tulojen summan ja syöttää kokonaissyötteen aktivointifunktiolle, joka päättää mitä neuroni välittää eteenpäin.

Yksittäinen neuroni ei pysty kovin monimutkaisiin tehtäviin, mutta yhdistelemällä neuroneita verkoksi on mahdollista valmistaa algoritmi huomattavasti haastavampiin tehtäviin.



Kuva 6: Kolmikerroksinen neuroverkko.

Kuvassa 6 on esitetty kolmikerroksinen neuroverkko. Jokaisen kerroksen väliset yhteydet sisältävät oppimisvaiheessa valmistellut painot, jotka vaikuttavat edellisen kerroksen syötteisiin. Kun tarkastellaan käsinkirjoitettujen numeroiden tunnistusta neuroverkon avulla, neuroverkkoon saapuvat syötteet ovat joko suoraan yksittäisiä pikseleitä, tai sitten piirreirrotuksessa mitattuja ominaisuuksia. Syötekerros välittää nämä syötteet piilokerrokselle, joka painojen ja aktivointifunktioiden perusteella välittää niitä eteenpäin tuloskerroksella. Tuloskerroksella on 10 solmua, yksi jokaiselle numerolle, ja se solmu, joka saa vahvimman signaalin aiemmalta kerrokselta, on neuroverkon lajittelema luokka.

3.3.1. Oppimisvaihe

Oppimisvaiheen aluksi verkon painot alustetaan satunnaisiksi arvoiksi, ja oppimisvaiheen tarkoituksena on säätää näitä painoja kohti optimaalista järjestystä. LeCun ja muut [1998] ovat kuvanneet neuroverkkoa funktiona $F(Z^p, W)$, jossa Z^p on syöte numero p ja W verkon sisältämät painot. Numeroiden luokittelutehtävässä tämän funktio syöte kuvautuu luokaksi, eli numeroksi, joka kyseinen syöte verkon mukaan on. Tässä vaiheessa algoritmille syötetään opetusdataa, josta algoritmi pystyy päättelemään, kuinka lähellä tai kaukana oikeasta vastauksesta sen arvio oli. Kullekin syötteelle voidaan laskea virhettä kuvaava funktio $E^p = C(T^p, F(Z^p, W))$, jossa funktio C laskee virheen tavoitellun tuloksen ja funktiosta F saadun tuloksen välillä. Oppimisvaiheen tavoitteena on minimoida tämä virhe kaikilla syötteillä, ja tämä onnistuu muuttamalla neuronien painoja. Yksi tapa laskea muutokset neuronien painoihin virheen perusteella on niin kutsuttu peruutusalgoritmi, joka laskee painomuutokset jokaiselle kerrokselle verkon tuloskerroksesta kohti alkupäätä peruuttaen.

3.3.2. Testausvaihe

Testausvaiheessa neuroverkolle annetut syötteet lasketaan neuronien painojen ja aktivointifunktioiden avulla neuroverkon läpi, ja tuloksena saadaan tuloskerrokselta arvio siitä, mistä numerosta on neuroverkon mukaan kyse. Neuroverkkojen monimutkaistuessa, eli piilokerrosten ja neuronien määrän kasvaessa, sekä opetus- että testausvaiheen vaatima laskentateho kasvaa jatkuvasti suuremmaksi. Laskentatehon lisääntyessä on siis mahdollista kehittää entistä tarkempia neuroverkkoja, ja hyödyntää esimerkiksi konvoluutioneuroverkkoja, joilla on jo saavutettu hyviä tuloksia myös muunlaisissa kuvientunnistustehtävissä [LeCun *et al.* 1998].

4. Yhdistetyt algoritmit

Yhdistetyt algoritmit on jaettu tätä tutkielmaa varten rinnakkaisiin ja sarjaan yhdistettyihin algoritmeihin, joilla molemmilla tavoitellaan parannuksia algoritmien suorituskykyyn.

Huang ja Suen [1995] esittävät, että yksittäisellä lajittelualgoritmilla ei yleensä päästä 100 % tunnistustarkkuuteen hahmontunnistuksessa, kuten numeroiden tunnistuksessa. Tästä syystä useilla eri lajittelualgoritmeilla tulisi päästä yksittäistä parempaan tunnistustarkkuuteen. Syynä tähän on se, että algoritmien ryhmäpäätökset ovat keskimäärin parempia kuin yksittäisen yksilön päätökset, sillä hieman eri tavalla toimivien lajittelualgoritmien on mahdollista täydentää toisiaan. Tästä syystä on hyvä tarkastella myös yhdistettyjä lajittelualgoritmeja, jo olemassa olevien algoritmien parantamisen lisäksi.

Toinen mahdollinen hyöty, joka voidaan saavuttaa yhdistelemällä lajittelualgoritmeja, on parempi tunnistusnopeus. Yhdistämällä algoritmeja sarjaan nopeammasta tarkempaan, voidaan helposti tunnistettavissa olevat numerot tunnistaa nopeilla algoritmeilla ja vaikeampiin yksilöihin käyttää tarkempia ja raskaampia algoritmeja. Näin säästyy laskentatehoa ja aikaa, kun kaikkiin tapauksiin ei tarvitse käyttää tarkinta algoritmia. [Chellapilla *et al.* 2006]

4.1. Rinnakkaiset algoritmit

Yksinkertaisimmillaan rinnakkaiset algoritmit toimivat useiden eksperttien yhdistelmän (Combination of multiple experts (CME)) enemmistöäänestyksessä. Tässä mallissa useat lajittelualgoritmit arvioivat samaa syötettä, ja äänestävät sitä luokkaa, johon arvioivat syötteen kuuluvan. Syötteen luokaksi valitaan se luokka, joka saa ehdottoman enemmistön lajittelualgoritmien äänistä. Äänien mennessä tasan, syöte voidaan joko hylätä, tai lajitella se toiseen ehdotetuista luokista. [Huang and Suen 1995; De Stefano *et al.* 2005]

Jo yksinkertaiselle enemmistöäänestyksellä on mahdollista parantaa tunnistustulosta, mutta menetelmän suurin haittapuoli on se, että kaikkia algoritmeja kohdellaan samanarvoisina [De Stefano *et al.* 2005]. Vaikka jokin algoritmeista olisi erittäin varma arviostaan, voivat muiden algoritmien arvaukset muuttaa CME:n päätöstä väärään suuntaan. De Stefano ja muiden [2005] mukaan yksi usein käytetty ratkaisu on painotettu enemmistöpäätös, jossa yksittäisten algoritmien äänet kerätään yhteen ja painotetaan kertoimilla ennen niiden summaamista yhteen päätöstä varten. Näin luotettavien algoritmien arvioilla on enemmän painoa lopputuloksessa, kuin epävarmempien algoritmien äänillä. Helppo ja usein käytetty menetelmä painojen, eli käytettävien kertoimien, laskentaan on testijoukon tunnistustaso eri algoritmeilla. Tässä tapauksessa, kun algoritmi äänestää jotain luokkaa kyseiselle syötteelle, sen äänelle asete-

taan kerroin sen mukaan, kuinka korkea tunnistusprosentti algoritmilla on ollut kyseiseen luokkaan kuuluvien näytteiden kanssa testiluokkaa testattaessa.

Testiluokan tuloksista lasketuilla painoilla on kuitenkin myös omat rajoitteensa. Niissä ei esimerkiksi pystytä hyödyntämään CME:n algoritmien tuottamaa informaatiota kokonaisuutena, vaan pyritään maksimoimaan yksittäisen algoritmin hyöty CME:lle. Painotettu enemmistöäänestys ei pysty esimerkiksi huomiomaan tilanteita, joissa algoritmien *a* ja *b* ollessa samaa mieltä syötteestä, ne ovat päätyneet aina oikeaan tulkintaan, mutta äänestyksen tulos on päätynyt virhetulkintaan muiden algoritmien äänien ja painojen takia. Älykkäämpi järjestelmä pystyisi huomioimaan esimerkiksi tämän kaltaisia yhdistelmiä äänestyksessä, ja näin ollen parantamaan tunnistustulosta entisestään. De Stefano ja muut [2005] tutkivat geneettisten algoritmien hyödyntämistä tähän tarkoitukseen. Tämän lisäksi esimerkiksi neuroverkko, joka saisi syötteenään CME:n äänet, olisi mielenkiintoinen tutkimuskohde tulevaisuudessa.

4.2. Sarjaan yhdistetyt algoritmit

Etsittäessä lajittelualgoritmia, joka pääsee parhaimpaan tunnistustarkkuuteen kaikissa mahdollisissa tilanteissa, päädytään usein tilanteeseen, jossa tarkin algoritmi on valitettavasti varsin raskas ja hidas [Chellapilla *et al.* 2006]. Esimerkiksi yllä esitelty algoritmien äänestysmalli on ymmärrettävästi selkeästi hitaampi kuin mikä tahansa malliin käytetty yksittäinen algoritmi, mikä saattaa aiheuttaa ongelmia suurien datamäärien tunnistuksessa, vaikka tunnistustarkkuus onkin parantunut. Eräs ratkaisu tasapainotteluun tarkkuuden ja tehokkuuden välillä on sarjaan kytketyt algoritmit.

Sarjaan yhdistettyjen algoritmien nopeus verrattuna esimerkiksi pelkkään CME:hen perustuu oletukseen, että analysoitavien syötteiden tunnistuksen haastavuus vaihtelee huomattavasti syötteestä toiseen. Tästä syystä esimerkiksi CME:n käyttö jokaisen syötteen kohdalla kuluttaa turhaan laskentatehoa ja aikaa, kun huomattavasti yksinkertaisempikin lajittelualgoritmi kykenisi tunnistamaan selkeimmät syötteet. Ratkaisuna voidaan siis ketjuttaa algoritmeja alkaen nopeimmasta ja päättyen tarkimpaan algoritmiin, jolloin haastavammat syötteet päätyvät loppujen lopuksi tarkimman algoritmin arviotavaksi, kun taas selkeämmät syötteet lajitellaan jo nopeimmassa algoritmissa.

Chellapilla ja muut [2006] käyttivät sarjaan yhdistettyinä algoritmeina konvoluutioneuroverkkoja, hyvin yksinkertaisista neuroverkoista erittäin syviin ja laajoihin neuroverkkoihin. Heidän algoritmiketjussaan nopein algoritmi oli testattaessa 41.6 kertaa nopeampi kuin monimutkaisin algoritmi, mutta nopeimman algoritmin virheprosentti oli 26.52 %, kun taas tarkin algoritmi ylsi vain 1.19 % virheprosenttiin. Toteuttamassaan empiirisessä kokeessa hyödyntäen näitä sarjaan yhdistettyjä algoritmeja, he ylsivät neljä kertaa nopeampaan

tunnistukseen verrattuna hitaimpaan algoritmiin, ilman että virheiden määrä kasvoi tunnistuksessa lainkaan. Kasvattamalla sallittujen virheiden määrää prosenttiyksiköllä heidän kokeessaan päästiin 15 kertaiseen nopeuteen verrattuna vain tarkimman algoritmin käyttöön.

5. Yhteenveto

Kuvadatan tulkinta on ollut jo pitkään - ja on todennäköisesti myös tulevaisuudessa - kiinnostava ja alati kehittyvä sektori tietojenkäsittelytieteen alalla. Tietokoneiden työpanoksella on mahdollista tehostaa tehtyä työtä monella eri alalla ja konenäkö on keskeisessä roolissa robottien omaksuessa entistä haastavampia näkö-raaja-koordinaatiota vaativia tehtäviä. Tässä tutkielmassa käsitelty numeroiden tunnistus on vielä suhteellisen yksinkertainen ja helposti määriteltävissä oleva ongelma verrattuna esimerkiksi valokuvissa olevia esineitä tai asioita lajitteleviin algoritmeihin, mutta yksinkertaisuudessaan se on tarjonnut hyvän mahdollisuuden tutkia erilaisia koneoppimisalgoritmeja ja niiden yhdistämisestä saatavia hyötyjä.

Kaikilla tässä tutkielmassa käsitellyillä lajittelualgoritmeilla on mahdollista saavuttaa tässä kyseisessä tehtävässä huipputasoa olevia tunnistusprosentteja, ja yhdistämällä useita algoritmeja voidaan sekä tehostaa prosessia, että pienentää virheiden määrä mahdollisimman vähäiseksi. Siirryttäessä muihin mahdollisiin tunnistustehtäviin on hyvä tutkia kriittisesti minkälaisia tarpeita kyseinen tehtävä pitää sisällään, ja sen perusteella päättää mikä tässä tutkielmassa esitellyistä algoritmeista sopii tehtävään parhaiten, tai tarvitseeko turvautua johonkin algoritmiin, jota tässä tutkielmassa ei ole esitelty.

Liun ja Fujisawan [2005] mukaan statistiikkaan perustuvat menetelmät, kuten kNN, ovat yleensä parempia yleistämään pienestä oppimisjoukosta saatua tietoa tunnistukseen kuin SVM ja neuroverkot. Oppimisjoukon kasvaessa taas SVM ja neuroverkot pystyvät molemmat selkeästi parempaan tunnistusprosenttiin ja suorituskykyyn kuin kNN. Yhdistelemällä algoritmeja voidaan siis hyödyntää erilaisten algoritmien hyvät puolet, ja tukea niitä osioita, joissa toiset algoritmit eivät pysty yhtä suorituskykyiseen tunnistukseen. Näin saavutetaan sekä parempia tunnistusprosentteja että tehokkaampia algoritmeja, kun raskaampia algoritmeja ei tarvitse käyttää kaikkiin tapauksiin. Koneoppimisalgoritmien koko potentiaali tulee testatuksi laskentatehon ja tietämyksen karttuessa tulevaisuudessa.

Viiteluettelo

Babu, U. R., Venkateswarlu, Y. and Chintha, A.K. 2014. Handwritten Digit Recognition Using K-Nearest Neighbour Classifier. In: *Proc. of World Congress on Computing and Communication Technologies (WCCCT).* 60-65.

Caesar, T. , Gloger, J. and Mandler, E. 1993. Preprocessing and feature extraction for a handwriting recognition system. In: *Proc. of the Second International Conference on Document Analysis and Recognition*. 408-411.

Chellapilla, K., Shilman, M. and Simard P. 2006. Combining multiple classifiers for faster optical character recognition. In: *Proc. of the 7th international conference on Document Analysis Systems*. 358-367

De Stefano, C., Della Cioppa, A. and Marcelli, A. 2002. An adaptive weighted majority vote rule for combining multiple classifiers. In: *Proc of the 16th International Conference on Pattern Recognition*. 192-195

Huang, Y. S. and Suen, C. Y. 1995. A method of combining multiple experts for the recognition of unconstrained handwritten numerals. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence.* 17, 1, 90-94.

Korakakis, M., Magkos, E. and Mylonas, P. 2014. Automated CAPTCHA Solving: An Empirical Comparison of Selected Techniques. In: *Proc. of the 9th International Workshop on Semantic and Social Media Adaptation and Personalization.* 44-47.

Lauer, F., Suen, Y. C. and Bloch, G. 2007. A trainable feature extractor for handwritten digit recognition. *Pattern Recognition*, 40, 6, 1816-1824.

LeCun, Y., Bottou, L., Bengio, Y. and Haffner P. 1998 Gradient-based learning applied to document recognition. In: *Proc. of the IEEE*, 86, 11, 2278-2324.

LeCun, Y. & Cortes, C. The MNIST Database. http://yann.lecun.com/exdb/mnist/. Checked: 29.1.2016.

Liu, C. L. & Fujisawa, H. 2005. Classification and learning for character recognition: comparison of methods and remaining problems. In: *International Workshop on Neural Networks and Learning in Document Analysis and Recognition.*

Matan, O., Baird, H., Bromley, J., Burges, C., Denker, J., Jackel, L., Le Cun, Y., Pednault, E., Satterfield, W., Stenard, C. and Thompson, T. 1992. Reading handwritten digits: a ZIP code recognition system. *Computer* 25, 7, 59-63.

Pesch, H., Hamdani, M., Forster, J. and Ney, H. 2012. Analysis of Preprocessing Techniques for Latin Handwriting Recognition. In: *Proc. of the 2012 International Conference on Frontiers in Handwriting Recognition*. 280-284.

Schölkopf, S. P., Vapnik, V., and Smola, A. J. 1997. Improving the accuracy and speed of support vector machines. In: *Advances in Neural Information Processing Systems*, *9*, 375-381.

Tuba, E. & Bacanin, N. 2015. An algorithm for handwritten digit recognition using projection histograms and SVM classifier. In: *Telecommunications Forum Telfor (TELFOR)*. 464-467

Weinberger, K. Q., Blitzer, J. and Saul, L. K. 2005. Distance metric learning for large margin nearest neighbor classification. In: *Advances in neural information processing systems*. 1473-1480.

Kuvat:

Kuva 3:
http://docs.opencv.org/3.0-beta/doc/tutorials/ml/introduction_to_svm/introduction_to_svm.html

Kuva 4: http://www.sbaban.org/tag/ml/

Kuva 5:
https://en.wikibooks.org/wiki/Artificial_Neural_Networks/Print_Version

Kuva 6: http://www.hindawi.com/journals/aai/2011/686258/fig1/